Efficient elliptic solvers for higher order DG discretisations on modern architectures and applications in atmospheric modelling

Betteridge, Jack

Award date:
2020

Awarding institution:
University of Bath

Efficient elliptic solvers for higher order DG discretisations on modern architectures and applications in atmospheric modelling

Jack Betteridge

Thesis submitted for the degree of Doctor of Philosophy

University of Bath

Department of Mathematical Sciences

November 2019

COPYRIGHT

LICENSE

DECLARATION OF ANY PREVIOUS SUBMISSION OF THE WORK

The material presented here for examination for the award of a higher degree by research has not been incorporated into a submission for another degree.

Candidate's signature .....

DECLARATION OF AUTHORSHIP

I am the author of this thesis, and the work described therein was carried out by myself personally, without exception.

Candidate's signature .....

SUMMARY

For problems in Numerical Weather Prediction (NWP), the time to produce a solution is a critical factor. Semi-implicit timestepping methods can speed up geophysical fluid dynamics simulations by taking larger timesteps than explicit methods. This is possible because they treat the fast (but physically less energetic) waves implicitly, and the timestep size is not restricted by the CFL condition for these waves. One disadvantage of this method is that an expensive linear solve must be performed at every timestep. However, using an effective preconditioner for an iterative method significantly reduces the computational cost of this solve, making a semi-implicit scheme faster overall.

Higher order Discontinuous Galerkin (DG) methods are known for having high arithmetic intensity making them well suited for modern HPC hardware. For smooth solutions higher order DG methods can be particularly efficient since errors decrease with a power of the polynomial degree. However, the linear problems which arise in semi-implicit timesteppers if DG methods are used for the spatial discretisation are difficult to precondition due to the large number of coupled degrees of freedom. This coupling arises since the numerical flux introduces off diagonal artificial diffusion terms. Those terms would result in a dense operator if the standard Schur complement reduction to an elliptic system is used.

In this thesis we use a hybridised DG (HDG) method to eliminate the original coupling and instead couple the system of equations to a sparse operator on the trace space, which is easier to precondition. This is achieved by considering the numerical flux variables which only lie on the facets of the mesh. Recent work by Kang, Giraldo and Bui-Thanh[34] solves the resulting system with a direct method. However, this becomes impractical for high resolution simulations due to the cost of this direct solve. Instead, in this thesis, we solve the resulting system using a non-nested geometric multigrid technique.

In this thesis we discretise and solve the non-linear shallow water equations, an important model system in geophysical fluid dynamics, using both DG and HDG methods. We develop a bespoke non-nested multigrid preconditioner based on work by Gopalakrishnan and Tan [31] and implement it using Firedrake, a Python framework for solving finite element problems via code generation. Hybridisation is performed using the Slate language, which is a part of Firedrake. We demonstrate the effectiveness of our hybridised DG scheme with non-nested multigrid preconditioner for a range of semi-implicit IMEX timesteppers and show these provide significant improvement over traditional DG methods with the same timestepping.

ACKNOWLEDGEMENTS

I would like to thank my supervisors Eike Müller and Ivan Graham for all of their help and advice. Both have always made time to meet and discuss mathematics, to sit down and assist when things were not going well and to share in excitement when there were new results. Their patience and guidance, through helping debug code, working through technical mathematics or reading over drafts of this thesis has always been appreciated. The support they have provided has been invaluable throughout my PhD, there is no way I could have achieved any of this without them.

I am grateful to the Bath Maths Department, I have had many opportunities to work with so many great people. I would also like to thank SAMBa (the EPSRC Centre for Doctoral Training in Statistical Applied Mathematics), specifically Susie Douglas (Centre Co-Director) and Jess Ohren (Centre Coordinator) for their work running the CDT that I have had the pleasure of being a part of. Their office has been on the same floor as mine for my entire time at Bath and they were always happy for me to stick my head in to ask questions or even just have a chat.

The PhD community at Bath has provided camaraderie, for which I have been very thankful. Events such as regular Wednesday cake, the postgraduate seminar series and Drinks in the Parade on Fridays kept me going through my four years here.

This research has made use of the HPC facilities at Bath and I am thankful for Stephen Chapman and Roshan Matthew for their HPC support. I would also like to thank Bath's central Research Software Engineer, James Grant, who has given me programming advice and provided many opportunities for me to use my programming abilities outside of my PhD project. Our informal meetings have often given me a different perspective on my work and its applications.

Outside of Bath, I would like to thank all of the Firedrake developers, who are spread over the globe, but special thanks to Colin Cotter, David Ham and Lawrence Mitchell. Thomas Gibson also deserves a special mention, his in depth knowledge of Slate and Firedrake, from his own PhD research, alongside his close collaboration in my final year of research has been invaluable.

The author (Jack Betteridge) is supported by a scholarship from the EPSRC Centre for Doctoral Training in Statistical Applied Mathematics at Bath (SAMBa), under the project EP/L015684/1.

This research made use of the Balena High Performance Computing (HPC) Service at the University of Bath.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF CODE LISTINGS

ABBREVIATIONS

AMG	Algebraic Multi-Grid.
CG	Conjugate Gradient.
CPU	Central Processing Unit.
DG	Discontinuous Galerkin.
DOF	Degree(s) of Freedom.
GMG	Geometric Multi-Grid.
HDG	Hybridised Discontinuous Galerkin.
HPC	High Performance Computing.
IMEX	IMplicit EXplicit.
KSP	Krylov Subspace (solver) *PETSc terminology*.
NWP	Numerical Weather Prediction.
PC	Preconditioner *PETSc terminology*.
PDE	Partial Differential Equation.
RAM	Random Access Memory.
SPD	Symmetric Positive Definite.
SWE	Shallow Water Equations.

NOTATION

$\boldsymbol{u}, \boldsymbol{v}$	A bold face vector is a vector valued function $\boldsymbol{u} : \Omega \to \mathbb{R}^n$.
$\underline{\phi}, \underline{u}$	A single underlined vector is as DOF vector.
$A, B, C, \ldots$	A Roman capitalised letter is often a matrix.
$\underline{\underline{\mathrm{A}}}, \underline{\underline{\mathrm{B}}}, \underline{\underline{\mathrm{C}}}, \ldots$	An upright double underlined Roman capitalised letter can also represent a matrix (usually larger and having block structure).
$\boldsymbol{a} \cdot \boldsymbol{b}$	The dot product of two vectors $\boldsymbol{a}^\top \boldsymbol{b}$ or $\sum_i a_i b_i$.
$\boldsymbol{a} \times \boldsymbol{b}$	The cross product of two vectors.
$\Phi_h \times \mathcal{U}_h$	Cross can also be the Cartesian product of two sets.
$\boldsymbol{a} \otimes \boldsymbol{b}$	The outer or tensor product of two vectors $\boldsymbol{a}\boldsymbol{b}^\top$.
$A : B$	The double dot product of two matrices $\sum_{i,j} A_{ij} B_{ij}$.
$A \oplus B$	The direct sum of two matrices $\begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix}$.

CHAPTER 1
INTRODUCTION

Solving PDEs in modern applications demands high accuracy, but computer processor speed[1] has not significantly increased in the past 20 years. An example of one demanding application is Numerical Weather Prediction (NWP), which is our focus.

At the heart of NWP are the Navier-Stokes equations, a system of non-linear partial differential equations, which model the motion of fluids. The Navier-Stokes equations need to be solved in a very short period of time in NWP (roughly 20 minutes [4]) to get a weather forecast in time. In the case of the atmosphere, the fluid of interest is the air. Whilst analytically intractable, it is still possible to find numerical solutions to Navier-Stokes, but this can only be done to a finite precision and is still very challenging. Solutions contain fine details, and to be fully resolved in a numerical method we typically require a fine mesh. One method of solving the system that arises is the Discontinuous Galerkin (DG) method, which is a discretisation method that leads to a system of non-linear equations. At the same time, to predict the weather, the atmosphere of the whole globe must be modelled. This combination of fine mesh over a large area leads to a huge system of non-linear equations to be solved.

The DG method approximates the solution to a PDE by some (possibly large degree) polynomial on each cell and allows discontinuities between cells in the mesh. The discontinuities in the DG method are small and disappear in the limit as the mesh is refined, because the solution of the PDEs that we solve is continuous.

1.1 Motivation

The Navier-Stokes equations are impossible to solve analytically in most domains, so must be discretised and the large system of equations solved on a modern high performance computer. This thesis includes a detailed investigation into Discontinuous Galerkin discretisation methods and their properties, and why they are well suited to modern High Performance Computing (HPC) architectures. The core achievement of this thesis is to construct an efficient solver for a DG discretisation and use this to solve the shallow water equations.

Current methods in use by MetOffice use a model with approximately 10^9 degrees of freedom, which correspond to a mesh with 10km grid spacing over the globe [4]. For predicting the weather, we want to know the values of these degrees of freedom as time

[1] single core clock speed, to be precise

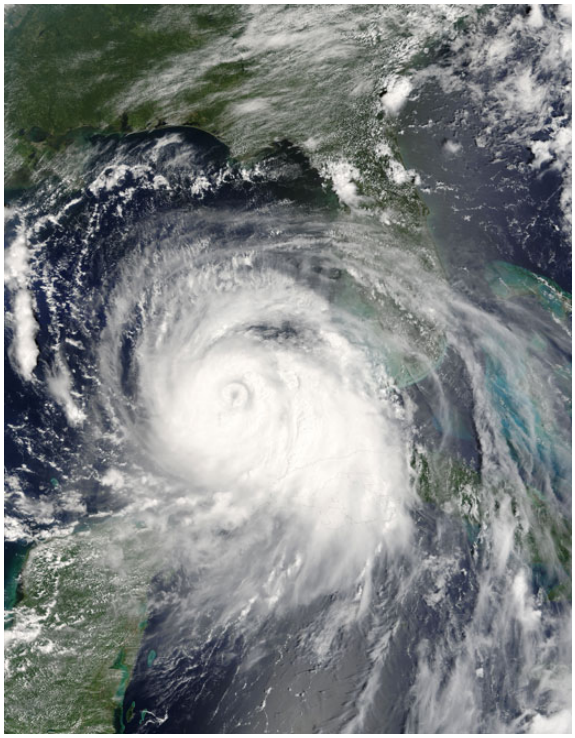

Figure 1-1: Satellite photograph of hurricane Katrina [3]

evolves, and this can be done by using a finite dimensional approximation to the model equations and using a timestepping method. Next generation weather models will use even more degrees of freedom.

Although DG methods have been known and studied since the 1970s, modern interest has been fuelled by properties that make them readily more efficient on modern architectures when compared to other standard methods. One attractive property is the high arithmetic intensity of the method when using higher order approximations and sum factorisation techniques. This is advantageous, since calculations can be performed faster than memory can be accessed on modern computers. While one value is being read from memory, approximately ten floating point operations can be performed by the CPU.

DG has many advantages over more traditional methods: The DG method can be applied on unstructured grids, which is an advantage over finite volume methods. It is naturally conservative, meaning physical quantities like mass, energy and momentum are conserved, which is an advantage over conforming finite element methods. Both the finite volume and finite element methods can overcome these issues, but require significant modification. At higher order (large polynomial degree), the DG method has a high rate of convergence, which typically grows as the polynomial degree used to approximate the solution on one mesh cell increases. To obtain this rate of convergence the solution has to be suitably smooth.

In this thesis we combine a DG spatial discretisation with advanced timesteppers to evolve the numerical solution forward in time. We will focus on IMplicit-EXplicit (IMEX) timesteppers which treat different parts of the underlying system differently. For instance different components of waves in the solution to the Navier-Stokes equations travel at different speeds, fast waves travel at approximately the speed of sound (300ms^{-1}) and slow waves travel at roughly 30ms^{-1}. Using an IMEX timestepper the fast waves can be treated implicitly. The physically irrelevant parts of the solution are stabilised, whilst the

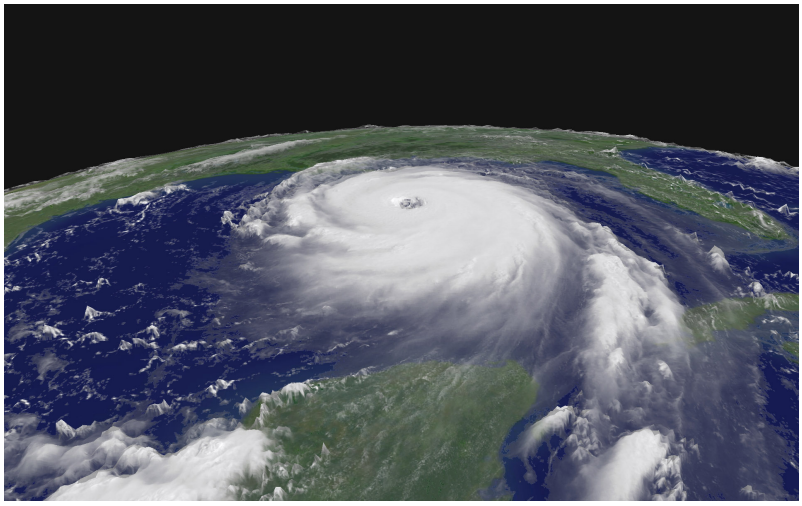

Figure 1-2: Photo-realistic simulation of a hurricane [5]

slow waves which are used to predict the movement of weather patterns and are treated explicitly. IMEX methods are efficient for current NWP models, but until now have been considered too expensive when combined with a DG discretisation.

Whilst an IMEX method allows for a larger timestep, because it stabilises fast waves, the method also requires the expensive solve of a large sparse system. Using a traditional Schur complement factorisation is not an effective way of solving the system either. It is an open question as to whether hybridisation is an efficient solver in this case. In this thesis we will construct an efficient solver for the hybridised DG method and apply this to an IMEX timestepper, thereby demonstrating the viability of semi-implicit timesteppers.

The thesis focuses on the solution of the Shallow Water Equations (SWE), which also exhibit a separation of fast and slow waves. For the SWE discretised using DG, we shall construct the matrix equations that when solved yield the numerical solution to the SWE. Solving the resulting system of equations directly would be time consuming and inefficient, hence iterative solvers are employed. The only feasible way to execute iterative solvers for a system of equations of this size on an operational timescale is by using modern High Performance Computing (HPC) architectures. For the MetOffice, where certain forecasts are run hourly, an operational timescale may be as short as 20 minutes to produce a 12 hour UK weather forecast [4]. If a forecast takes longer than this, the weather that is being predicted will already start happening, making the prediction less valuable.

The power of modern supercomputers comes from connecting many multi-core processors together in parallel. But, at the dawn of exascale computing, we are becoming increasingly aware that current methods for solving numerical problems may not be taking full advantage of the available hardware.

1.2 Main achievements

The content of chapter 2 outlines the physical properties of the Shallow Water Equations (SWE) which are the model system of equations that we consider. The shallow water equations are often used as a starting point for studying performance of solvers for NWP, this is because they are simpler than the full Navier-Stokes or compressible Euler equations, but still maintain key features. One important feature is the separation of wave speeds into fast and slow waves. The shallow water equations are also interesting in their

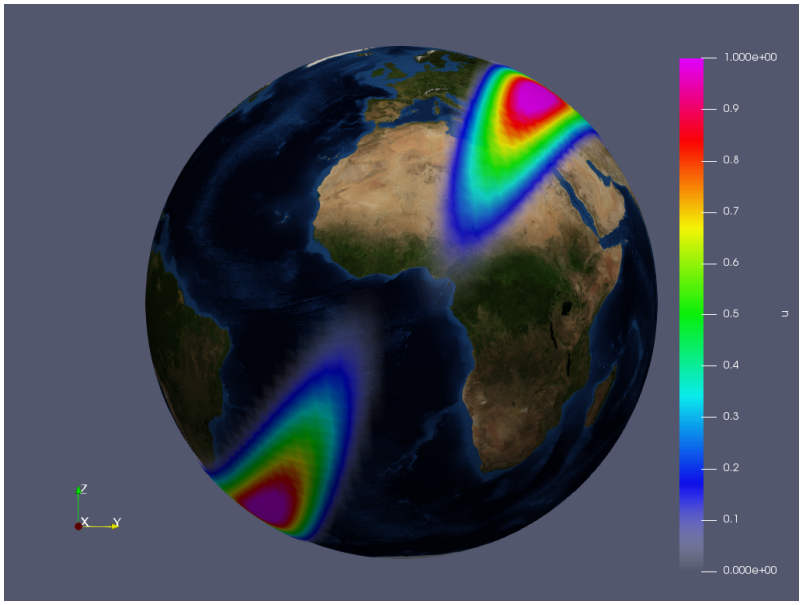

Figure 1-3: Simulation of weather patterns on the Earth

own right, and are frequently used in oceanographic modelling. In this chapter we find stationary solutions to the SWE that are used in the implementation (chapter 7), as well as demonstrate how more traditional finite difference and conforming finite element schemes can be constructed from the continuum equations.

In chapter 3 we investigate the DG method in a general setting and the properties we have as a result of choosing a discontinuous discretisation. We compare the method to a conforming method and point out specific desirable properties that arise, specifically matrix structure that modern HPC architectures can take advantage of. We examine the DG discretisation for the advection problem to give a concrete example of the method. The same discretisation technique is then performed on the more complicated shallow water equations. This DG formulation of the SWE is then used as the basis for the investigation in rest of the thesis.

In chapter 4 we study the Hybridised Discontinuous Galerkin (HDG) method, which is a modification of the DG method. We take the shallow water equations defined in chapter 3 and use Lagrange multipliers to eliminate the coupled degrees of freedom to obtain a formulation defined only on the facets (element boundaries) that is less expensive to work with. Hybridisation overcomes the fact that DG numerical fluxes introduce additional coupled degrees of freedom. Whilst hybridisation of the SWE has been investigated before, we overcome the issues encountered by Kang et al. [34] by combining the hybridisation technique with multigrid solvers. This chapter also investigates suitable choices of timesteppers for our discretisation of the SWE. Examples of explicit timesteppers are given, which are already known to be efficient when combined with DG discretisations. We also give examples of IMEX timesteppers, utilising hybridisation and multigrid solvers, and we later show (in chapter 8) that the fastest of these are only 60% slower than similarly accurate explicit timesteppers for DG. Furthermore, these hybridised formulations of the DG method execute roughly twice as fast as the same method using a standard DG discretisation.

Chapter 5 investigates how the resulting system of linear equations can be solved and we give examples of some appropriate choices for iterative methods. The convergence of iterative methods can be accelerated by using an appropriate preconditioner. In this

chapter we prove the effectiveness of an approximate Schur complement preconditioner for lowest order DG methods. We discuss how hybridisation can be used as an effective preconditioner that works at higher order and reduces the large system of linear equations to a much smaller elliptic problem. Different powerful multigrid techniques are explored at the end of the chapter as these are incredibly efficient solvers for elliptic problems. We combine these three components into a new efficient solver for the shallow water equations.

There are complex challenges associated with applying multigrid to a hybridised system. In chapter 6 a new non-nested multigrid method for the hybridised SWE system is developed to overcome these challenges. Although such techniques have been proposed before for the Poisson equation [31], this thesis provides a novel extension to the linearised SWE.

Chapter 7 describes in detail the algorithms and implementation of the solver. We use the DG discretisation from chapter 3 and solver techniques from chapter 5 to construct algorithms for solving the shallow water equations. The concepts of hybridisation from chapter 4 and non-nested multigrid from chapter 6 are then combined together to produce new effective, scalable and efficient preconditioners for the the method. We implement our own solvers and preconditioners in the Firedrake [44] framework which supports discontinuous function spaces, generate the finite element matrices in PETSc [11] and perform hybridisation using Slate [29]. This combined with IMEX timsteppers allows us to effectively solve the equations in parallel using a modern HPC facility. This new preconditioner for HDG discretisations combined with IMEX timestepping are offeres significant performance gain over DG methods with same timesteppers. The new preconditioner is only 60% slower than similarly accurate explicit DG timesteppers.

1.3 Computational results

We show in chapter 8 that our preconditioner, which combines hybridisation and non-nested multigrid is better than other preconditioners over a range of different problems. The main computational investigation compares different preconditioners to this hybridised multigrid preconditioner for both explicit and IMEX timesteppers. Our preconditioner shows significant improvements against other existing DG IMEX methods. We have also successfully developed a preconditioner for HDG IMEX that is only 60% slower than explicitly timestepped DG methods.

Having outlined the algorithms and implementation, in chapter 8 we present the numerical results obtained from the DG discretisation of the SWE. We compare both linear and non-linear shallow water equations to see the different aspects of solving a simplified problem when compared solving a problem with more realistic set up. We use the range of different timesteppers from chapter 4 to assess which are most effective. The code is also run with increasingly large problems on different numbers of processors and we observe near perfect strong *and* weak scalability of our preconditioner for high order DG discretisations.

Chapter 9 offers some conclusions of our work. The overarching achievement of this thesis is the development of efficient solvers and preconditioners for the DG method for atmospheric modelling and their efficient implementation on modern HPC architectures.

1.4 Dissemination

Over the past three years the research presented here has also been presented at a wide range of conferences.
Use of high performance computing was presented at:

- **HPC symposium 17, 18, 19** *University of Bath* Contributed talk, Poster, Contributed talk (respectively)
- **HPC champions 19** *University of Birmingham* Invited talk

Use and development of new software frameworks for numerically solving PDEs was presented at:

- **Firedrake 18** *Imperial College London* Contributed talk
- **PDESoft 18** *Bergen, Norway* Poster
- `Firedrake.GTMGPC` Preconditioner implemented and included as part of Firedrake (joint work with Thomas Gibson)
- **Firedrake 19** *Durham University* Contributed talk

Large international mathematics conferences focused on different areas of numerical analysis:

- **SciCADE 17** *University of Bath* Poster
- **19th Copper Mountain conference on multigrid methods (2019)** *Copper Mountain, Colorado, USA* Contributed talk

Future endeavours:

- **SIAM Conference on Parallel Processing for Scientific Computing (PP20)** *Seattle, Washington, USA* Invited minisymposium talk
- Paper in preparation entitled "Multigrid preconditioners for the hybridized Discontinuous Galerkin discretisation of the shallow water equations" with coauthors Thomas Gibson, Ivan Graham and Eike Müller

CHAPTER 2
SHALLOW WATER EQUATIONS

To simulate the atmosphere and predict the weather, the compressible Navier-Stokes equations can be solved numerically. One of the key features of the Navier-Stokes equations in this application is the separation of fast and slow waves.

The Navier-Stokes system of coupled hyperbolic PDEs is difficult to solve even numerically, due to its complexity. For this thesis our main focus will be solving the shallow water equations which can be obtained by integrating out the z dimension of the Navier-Stokes equations for fluid dynamics.

The shallow water equations (SWE) describe how a body of water, which has a depth much smaller than the horizontal length, evolves in time. Such a model is suitable for studying waves in the oceans, where the depth is a few kilometres, but horizontal extent is several thousands of kilometres. We can include in the shallow water equations terms for the bathymetry (the shape of the ocean floor below the surface of the water), and Coriolis terms (which typically occur on Earth as a result of its rotation on its axis).

The SWE are a suitable model system since they also show the separation between fast and slow waves previously mentioned for the Navier-Stokes equations. This separation is shown in section 2.2.

2.1 Non-linear SWE

The fully non-linear shallow water equations in two spatial dimensions, defined on some region $\Omega \subset \mathbb{R}^2$, for some time interval $[0, T_0]$, are:

$$\frac{\partial \phi_S}{\partial t} + \nabla \cdot \boldsymbol{U} = 0 \tag{2.1}$$

$$\frac{\partial \boldsymbol{U}}{\partial t} + \nabla \cdot \left[\frac{\boldsymbol{U} \otimes \boldsymbol{U}}{\phi_B + \phi_S} + \left(\frac{1}{2}\phi_S^2 + \phi_B \phi_S \right) \mathcal{I}_2 \right] = -f(\hat{\boldsymbol{k}} \times \boldsymbol{U}) + \phi_S \nabla \phi_B \tag{2.2}$$

We will use the equations as they are defined in the paper by Giraldo and Restelli [30]. Boundary conditions and initial conditions still need to be stated to fully define the problem, these will be discussed later.

Equation (2.1) describes the conservation of mass and equation (2.2) describes the conservation of momentum in the system. Whilst physically these quantities have units, we will work with the non-dimensionalised equations (2.1) and (2.2).

Here $\phi_S : \Omega \times [0, T_0] \to \mathbb{R}$, is the height potential of the water. At a given point $\phi_S = gh_S$, where h_S is the height measured up from the mean water height $H \in \mathbb{R}$ and $g = 9.81\text{ms}^{-1}$ is acceleration due to gravity. $\phi_B : \Omega \to \mathbb{R}$ describes the bathymetry and $\phi_B = gh_B$. The quantity h_B is measured down from the mean water height and is assumed to be constant in time. The labelled diagram in figure 2-1 shows these quantities. $\nabla \cdot$ denotes the (where necessary, row-wise) divergence of a quantity. Explicitly, for a general vector valued function $\boldsymbol{f}$ and matrix function G:

$$\nabla \cdot \boldsymbol{f} = \begin{pmatrix} \partial_x \\ \partial_y \end{pmatrix} \cdot \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \partial_x f_1 + \partial_y f_2, \qquad \nabla \cdot G = \begin{pmatrix} \partial_x \\ \partial_y \end{pmatrix} \cdot \begin{pmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \\ G_{31} & G_{32} \end{pmatrix} = \begin{pmatrix} \partial_x G_{11} \\ \partial_x G_{21} \\ \partial_x G_{31} \end{pmatrix} + \begin{pmatrix} \partial_y G_{12} \\ \partial_y G_{22} \\ \partial_y G_{32} \end{pmatrix} \tag{2.3}$$

$\boldsymbol{U} : \Omega \times [0, T_0] \to \mathbb{R}^2$ is the momentum density, which we will simply refer to as momentum. The momentum density is $\boldsymbol{U} = (\phi_S + \phi_B)\boldsymbol{u}$, where $\boldsymbol{u}$ is the velocity at a given point. $\otimes$ denotes the tensor (outer) product of two vectors

$$\underbrace{\boldsymbol{a}}_{\in \mathbb{R}^m} \otimes \underbrace{\boldsymbol{b}}_{\in \mathbb{R}^n} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \otimes \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_m b_1 & a_m b_2 & \cdots & a_m b_n \end{pmatrix} \in \mathbb{R}^{m \times n},$$

and $\mathcal{I}_2$ denotes the 2×2 identity matrix.

$-f(\hat{\boldsymbol{k}} \times \boldsymbol{U})$ is the Coriolis term where $f : \Omega \to \mathbb{R}$ is scalar valued. $\hat{\boldsymbol{k}}$ is the unit normal facing out of the plane. Since we will be working in two dimensions in the plane we expand the vector to calculate the cross product

$$\hat{\boldsymbol{k}} \times \boldsymbol{U} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \times \begin{pmatrix} U_1 \\ U_2 \\ 0 \end{pmatrix} = \begin{pmatrix} -U_2 \\ U_1 \\ 0 \end{pmatrix}$$

and take the first two components. Some texts denote $\boldsymbol{U}^\perp = \hat{\boldsymbol{k}} \times \boldsymbol{U}$.

Note 2.1.1. We consider the shallow water equations (equations (2.1) and (2.2)) in the plane as a simplification. For atmospheric modelling we would need to pose these equations on the surface of a sphere embedded into $\mathbb{R}^3$, requiring the equations must be transformed onto the curved surface. Doing this the vector $\hat{\boldsymbol{k}}$ now varies over the domain, as does the tangent plane to the surface. When discretising the sphere, the cell normal (a term that appears in the equations for the DG flux, see chapter 3) no longer lies in the tangent plane and this must be corrected for a DG method to remain conservative. Posing the equations in the plane avoids this issue, so whilst it is possible to work with a spherical formulation of the SWE, in this thesis we do not.

Another important consideration the shallow water equations is the set of conserved quantities. The SWE in equations (2.1) and (2.2) are written in terms of the potential ϕ_S and momentum $\boldsymbol{U}$. Equation (2.1) ensures that mass is conserved, which is an essential property when performing shallow water simulations over long periods of time.

The second equation, equation (2.1) ensures that momentum is conserved. Both conservation of mass and momentum are enforced by the system equations, but it can be shown that the continuous equations conserve other important quantities.

Rossby shows[47] that the potential vorticity is also conserved. We define the relative vorticity, $\zeta := \partial_x u_2 - \partial_y u_1$, where u_1, u_2 are the components of the *velocity*, which is the

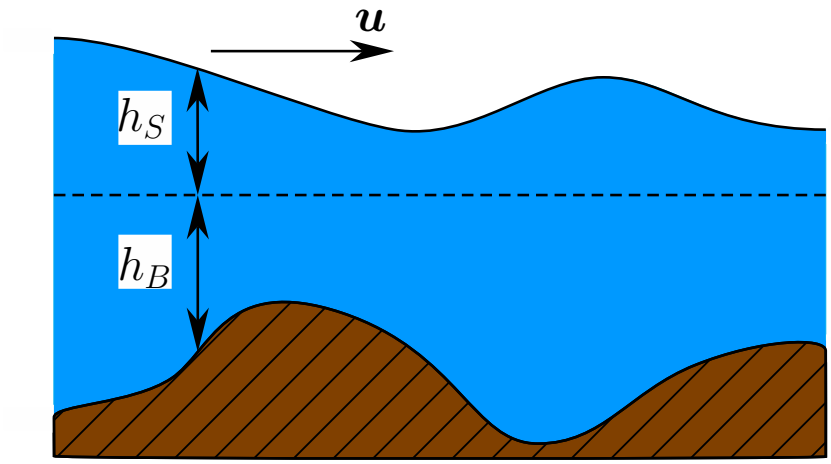

Figure 2-1: Diagram of variables in the SWE

two dimensional analogue of the three dimensional curl:

$$\nabla \times \boldsymbol{u} = \nabla \times \begin{pmatrix} u_1 \\ u_2 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \partial_x u_2 - \partial_y u_1 \end{pmatrix} = (\partial_x u_2 - \partial_y u_1)\hat{\boldsymbol{k}}$$

Taking the curl of equation (2.2) and simplifying we obtain

$$\frac{\partial}{\partial t}\left(\frac{\zeta + f}{h_s + h_B}\right) = 0,$$

where the term $\frac{\zeta+f}{h_s+h_B}$ is the potential vorticity for the SWE, which is conserved.

Potential vorticity is an essential quantity for applications in numerical weather prediction, as it is used as a tracer for fluid parcels through the simulation.

Energy is another physical quantity that is conserved in the continuous equations, but is less operationally useful. Operational models are often energy dissipative, and conservation of energy is only enforced on certain terms of the governing equations. We revisit these quantities in section 3.3 when considering a discretisation of the SWE.

2.2 Linearisation

We now turn our attention to a linearisation of the shallow water equations. A linear system of PDEs is less computationally expensive to solve than a non-linear system and importantly, the linearisation presented here contains the fast waves from the non-linear part of the system.

Solving the linearised SWE, we obtain an approximation to the non-linear solution, but we can also use the linearisation when we perform semi-implicit timestepping. The shallow water equations linearised around the "lake at rest", where $\phi_S = 0$, $\boldsymbol{U} = \boldsymbol{0}$, are given by:

$$\frac{\partial \phi_S}{\partial t} + \nabla \cdot \boldsymbol{U} = 0 \tag{2.4}$$

$$\frac{\partial \boldsymbol{U}}{\partial t} + \nabla(\phi_S \phi_B) = -f(\hat{\boldsymbol{k}} \times \boldsymbol{U}) + \phi_S \nabla \phi_B \tag{2.5}$$

These arise by keeping only leading order terms from the Taylor expansion of the non-linear SWE and removing all of the terms not linear in ϕ_S and $\boldsymbol{U}$. This leads to the linear operator L:

$$L\begin{pmatrix}\phi_S\\ \boldsymbol{U}\end{pmatrix} = -\begin{pmatrix}\nabla\cdot\boldsymbol{U}\\ \nabla(\phi_S\phi_B) + f(\hat{\boldsymbol{k}}\times\boldsymbol{U}) - \phi_S\nabla\phi_B\end{pmatrix}. \tag{2.6}$$

To show that we have a separation of wave speeds we start by simplifying equation (2.5). Using the product rule we get

$$\frac{\partial \boldsymbol{U}}{\partial t} + \phi_B\nabla\phi_S = -f(\hat{\boldsymbol{k}}\times\boldsymbol{U}). \tag{2.7}$$

If we have a divergence free flow ($\nabla\cdot\boldsymbol{U}=0$) in equation (2.4) and balance the potential with the Coriolis term,

$$\phi_B\nabla\phi_S^{\text{stat}} = -f(\hat{\boldsymbol{k}}\times\boldsymbol{U}^{\text{stat}}), \tag{2.8}$$

we can construct stationary solutions $\phi_S^{\text{stat}}, \boldsymbol{U}^{\text{stat}}$. Here $\boldsymbol{U}^{\text{stat}}$ is the momentum for the slow geostrophic waves. We show two solutions that satisfy equation (2.8) in section 2.3. This shows that the solution to the SWE contains slow waves when there is a Coriolis term present.

Note 2.2.1. The term *geostrophic* refers precisely to flow that satisfies equation (2.8). That is when the pressure gradient term (involving $\nabla\phi_S$) is balanced by the Coriolis term (involving f). The fluid in this case (not the wave) moves along a path that is parallel to lines of constant pressure gradient (often called isobars).

If we assume that the SWE as stated in equations (2.4) and (2.7) has wave-like solutions for the height potential and both components of the momentum:

$$\begin{aligned}\phi_S(t,\boldsymbol{x}) &\propto e^{i(\boldsymbol{p}\cdot\boldsymbol{x}-\omega t)},\\ \boldsymbol{U}_1(t,\boldsymbol{x}) &\propto e^{i(\boldsymbol{p}\cdot\boldsymbol{x}-\omega t)},\\ \boldsymbol{U}_2(t,\boldsymbol{x}) &\propto e^{i(\boldsymbol{p}\cdot\boldsymbol{x}-\omega t)}.\end{aligned} \tag{2.9}$$

Here i is the imaginary unit, $\boldsymbol{p}$ is the wave vector describing the direction in which the wave travels, $\boldsymbol{x}$ the spatial coordinate, t is time and ω is the wave frequency. We also make the simplifying assumption that the bathymetry is flat, that is ϕ_B is constant in space and time.

By substituting equation (2.9) into equations (2.4) and (2.7) and eliminating terms, we see that ω satisfies the dispersion relation

$$\omega\left(\omega^2 + f^2 - c_g^2|\boldsymbol{p}|^2\right) = 0, \qquad c_g^2 = \phi_B. \tag{2.10}$$

Here c_g is the maximum wave speed in the system.

There are two solutions here, first when $\omega = 0$. These solutions correspond to slow (Rossby) waves, which in this case do not propogate as we have a constant Coriolis parameter (f). This corresponds precisely to solutions that satisfy equation (2.8), and are what allow us to construct the stationary solution in section 2.3.

The other solution satisfies

$$\omega^2 = c_g^2|\boldsymbol{p}|^2 - f^2.$$

These are the fast (gravity) waves, which are irrotational ($\nabla\times\boldsymbol{U}$). The positive and negative solutions correspond to waves propagating in opposite directions. We are less interested in these as they carry little energy and can pollute our numerical solution. These

must be carefully balanced with the terms in the non-linear SWE to obtain physically realistic solutions.

We see that the linear operator L in equation (2.6) does indeed contain the terms that produce fast waves. When we later discretise the SWE in time we will treat the linear part (corresponding to L) implicitly, stabilising these fast waves and allowing larger timesteps to be taken.

2.3 Stationary vortex

Before we discuss methods for numerically solving the SWE, it is worth noting that for specific initial conditions we can solve the SWE analytically. In fact we have already used one analytic solution to construct the linearisation in section 2.3, namely the lake at rest.

If we start by considering a body of water that is completely flat, that is ϕ_S is constant, and at every point the water has no velocity ($\boldsymbol{U} = 0$), then we do not expect the water to move. This solution makes sense intuitively, but is confirmed by setting $\phi_S = C$ (some constant) and $\boldsymbol{U} = 0$. If this is the case we have $\partial_t \phi_S = 0$ and $\partial_t \boldsymbol{U} = 0$ and our solution is a stationary solution to the SWE (both the linear and non-linear SWE in this case).

$$\begin{aligned} \phi_S(\boldsymbol{x}, t) &= C \\ \boldsymbol{U}(\boldsymbol{x}, t) &= 0 \qquad \forall x \in \Omega \qquad \forall t \in [0, T] \end{aligned}$$

Here the term stationary means $\partial_t \phi_s = 0$ and $\partial_t \boldsymbol{U} = 0$ and so neither of the quantities ϕ_S nor $\boldsymbol{U}$ change value as time advances. The term stationary does **not** mean that the water being modelled does not move as the next solution demonstrates.

We will now consider a different stationary solution to the lake at rest, which we will refer to as the "stationary vortex". For this analytic solution we consider a body of water that is rotating around the centre of a periodic unit square domain at some fixed constant velocity. Since the flow is rotational we know that $\nabla \cdot \boldsymbol{U} = 0$ and $\partial_t \phi_S = 0$, so the solution is stationary in ϕ_S. In order to make the solution stationary in $\boldsymbol{U}$ we balance the height potential ϕ_S with the Coriolis parameter f in equation (2.2) or equation (2.5) so that $\partial_t \boldsymbol{U} = 0$.

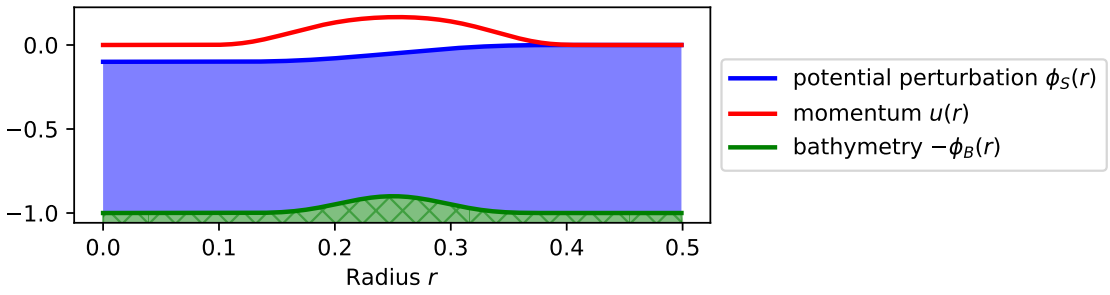

Figure 2-2: Profile of vortex, to scale

The stationary solution is radially symmetric and is naturally expressed in polar coordinates. The domain is divided into three parts, a circle of radius r_- centred in the middle of the domain where the height potential is constant, an annulus $r_- < r < r_+$ containing the banked sides of the vortex and finally the region $r > r_+$ where the height is again constant.

An exact solution of the SWEs is given by the following smooth function for the height potential:

$$\phi_S(r) = \begin{cases} -\Delta\phi_S & \text{for } r < r_- \\ -\frac{1}{2}\Delta\phi_S \left[1 + \tanh\left(\frac{\sigma}{r-r_-} + \frac{\sigma}{r-r_+}\right)\right] & \text{for } r_- \leq r \leq r_+ \\ 0 & \text{for } r_+ < r \end{cases} \tag{2.11}$$

Recall that the height is measured from the average water height *upwards*. Equation (2.11) is only a solution for a correctly chosen velocity, given in equation (2.13). Here σ is a parameter used to control how steep the sides of the vortex are.

We can also include bathymetry in the annular region

$$\phi_B(r) = \begin{cases} 1 - \Delta\phi_B \exp\left(\frac{1}{r-r_+} + \frac{4}{r_+-r_-} - \frac{1}{r-r_-}\right) & \text{for } r_- \leq r \leq r_+ \\ 1 & \text{otherwise.} \end{cases} \tag{2.12}$$

Recall that the bathymetry depth is measured from the average water height *downwards*, so $\phi_B = 1$ is one dimensionless height unit below the average height of the water.

The velocity is purely tangential and can be written as $\boldsymbol{u}(r) = u(r)\boldsymbol{e}_\theta$, where $\boldsymbol{e}_\theta$ is a unit vector in the tangential direction pointing anticlockwise. For the non-linear SWEs in equations (2.1) and (2.2) the function $u(r)$ is given by

$$u(r) = \frac{r}{2L}(\phi_S + \phi_B)\left(-1 + \sqrt{1 + 4L^2\frac{\phi_S'}{r}}\right), \tag{2.13}$$

where ϕ_S' is the derivative with respect to r of equation (2.11) and L is the length of the domain. The velocity in equation (2.13) is actually suitable for other choices of ϕ_S and ϕ_B, but equations (2.11) and (2.12) ensure $\phi_S, \phi_B \in C^\infty$.

For the linear SWEs in equations (2.4) and (2.5) the tangential velocity is

$$u(r) = L(\phi_S + \phi_B)\phi_S'.$$

The a graphical representation of the solution is shown in figure 2-3. The parameters used were $L = 1$, $r_- = 0.05$, $r_+ = 0.45$, $\sigma = 0.25$, $\Delta\phi_S = 0.1$, $\Delta\phi_B = 0.1$, but the vertical scale has been exaggerated to make the shape of the vortex clearer. Figure 2-2 gives an accurate and to scale plot of the radial profile of the vortex, in the non-linear case (the linear case is very similar).

By constructing an analytic solution we can verify numerical solutions to the SWE. Using the stationary vortex as a reference solution, we can calculate the numerical error by finding the difference between the numerical solution and the reference solution. Not only can this be a useful check that the numerical solution is correct, but can also be used to track the order of convergence.

2.4 Timestepping

One property of the Navier-Stokes equations is the separation of fast and slow waves. Slow waves, which travel at roughly $30ms^{-1}$, carry the information that we are most interested in, such as large scale weather patterns, advection and the effects of the Coriolis force. The fast waves are often called acoustic waves as they travel at approximately the speed of sound in air ($300ms^{-1}$), these waves are of little relevance to the dynamics as they carry

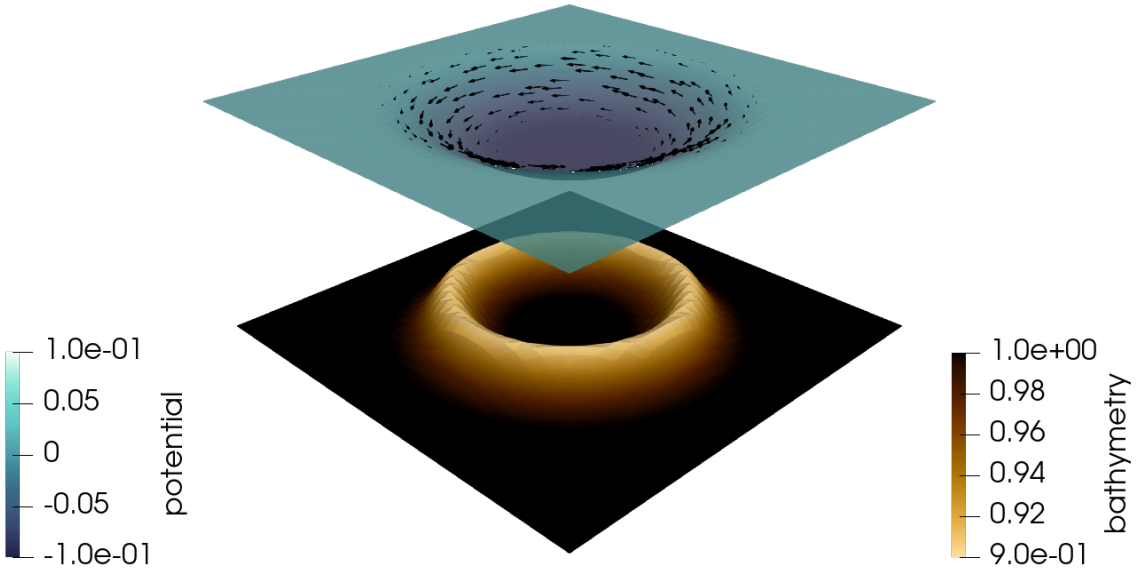

Figure 2-3: An example of the stationary vortex analytic solution to the SWE

only a small amount of energy. However, we cannot completely ignore the fast waves as they are essential for driving the non-linear dynamics in the Navier-Stokes equations. This suggests that we can treat the fast and slow waves differently when we solve the equations.

This separation of fast and slow waves is less well defined in the shallow water equations than in the Navier-Stokes equations, but we can still make a distinction between the different wave speeds. To see how this separation helps us solve the SWE more effectively. Consider the generic time dependent PDE

$$\frac{\partial q}{\partial t} = \mathcal{N}(q) + \mathcal{L}(q),$$

where $q(t)$ is the quantity of interest, $\mathcal{L}$ contains only terms that generate fast waves and $\mathcal{N}$ contains the remaining terms. To evolve this equation forward in time, numerically, we need a discretisation in time, which requires us to chose a timestepping method.

We will start by looking at the explicit Euler timestepping method. For this we approximate the time derivative and evaluate the non-linear operator using the solution at the current time step, denoted $q^{(n)}$.

$$\frac{q^{(n+1)} - q^{(n)}}{\Delta t} = \mathcal{N}(q^{(n)}) + \mathcal{L}(q^{(n)})$$

The solution at the next time step is obtained without solving a linear system, just evaluate the right hand side using $q^{(n)}$.

$$q^{(n+1)} = q^{(n)} + \Delta t[\mathcal{N}(q^{(n)}) + \mathcal{L}(q^{(n)})]$$

This method is very computationally cheap as no solves are required. The issue with this method is that it requires the time step Δt to be very small for the method to remain stable.

For a problem like the SWE, if the time step is chosen too large then fast waves become unstable and pollute the solution. The solution to this issue is to choose a time step that

is small enough that this phenomenon does not occur. The relation that Δt must satisfy is the CFL condition [22]:

$$C_{CFL} := u\frac{\Delta t}{\Delta x} < C \approx 1, \tag{2.14}$$

where Δx is the distance between adjacent points in the spatial disceretisation and u is the maximum wave speed in the system. We saw in equation (2.10) that for a linear problem $u = \sqrt{\phi_B}$. For explicit Euler the value of the constant C is approximately 1.

If for a quantity of interest $q(x,t)$, which now has some spatial dependence, we want to look at solutions with a high spatial resolution an explicit method restricts us to very small time steps. Since it is almost always desirable to look at higher resolutions, tiny timesteps mean that explicit Euler is not a suitable timestepping method. To get around this timestep restriction we could use an implicit method, for example implicit Euler. This differs from the explicit Euler method as we evaluate the non-linear operator at the *next* timestep.

$$q^{(n+1)} - \Delta t[\mathcal{N}(q^{(n+1)}) + \mathcal{L}(q^{(n+1)})] = q^{(n)}$$

To obtain the solution at the next time step we must invert the operator $(\mathcal{I} - \Delta t\mathcal{N} - \Delta t\mathcal{L})$, where here $\mathcal{I}$ just denotes the identity operator.

$$q^{(n+1)} = (\mathcal{I} - \Delta t\mathcal{N} - \Delta t\mathcal{L})^{-1} q^{(n)}$$

In this method fast waves are suppressed and do not grow, even if we take larger time steps. This is ideal, but now at every time step we have to invert a non-linear operator, which is *very* computationally expensive.

Looking at these two simple timestepping methods, we observe that the trade off for taking larger time steps is a method that is more computationally expensive to calculate the next time step. To obtain a compromise, we treat the fast and slow waves in the problem differently. We do this by splitting the non-linear operator into a non-linear part and a linear part. Note that the fast waves that arise in the problem are due to terms in $\mathcal{L}$. We use a semi-implicit timestepping method, that treats the linear part implicitly and the non-linear part explicitly.

$$\frac{q^{(n+1)} - q^{(n)}}{\Delta t} = \mathcal{N}(q^{(n)}) + \mathcal{L}(q^{(n+1)})$$

Equivalently, writing this as

$$q^{(n+1)} - \Delta t\mathcal{L}(q^{(n+1)}) = q^{(t)} + \Delta t\mathcal{N}(q^{(n)}),$$

we see that we must first evaluate the non-linear operator $(\mathcal{I} + \Delta t\mathcal{N})$ at the current timestep, which we know is computationally inexpensive. To obtain the solution at the next time step we also need to invert the *linear* operator $(\mathcal{I} - \Delta t\mathcal{L})$. Because the operator for the implicit part of the method is linear, the solve is far less computationally expensive than for a purely implicit method.

This semi-implicit method really is the best of both worlds as we can now take far larger time steps. In the context of the SWE the aim is to find an $\mathcal{L}$ that encapsulates the dynamics of the fast waves, but their contribution gets stabilised by treating $\mathcal{L}$ implicitly. The solve required at each time step now involves a linear operator, which is computationally cheaper than a solve involving a non-linear operator, but is still the most computationally expensive part of this method. The efficient treatment of the operator $(\mathcal{I} - \Delta t\mathcal{L})$ will be the focus of later chapters.

CHAPTER 3

THE DISCONTINUOUS GALERKIN METHOD

Discontinuous Galerkin (DG) methods were first studied in the 1970s [38, 46] in the context of the neutron transport problem, but recent interest for using DG for other problems is in part fuelled by their use in numerically solving PDEs on high performance computing (HPC) architectures. If a PDE is discretised using a DG method the resulting matrix has convenient block structure, despite having more degrees of freedom than a comparable conforming finite element method.

Like other finite element methods, the DG method can use any shape of cell and hence work over arbitrary geometries.

The block structure of the matrix is an advantage as it gives better data locality. That is, the matrices that act on the interior of a given cell in a mesh are precisely the blocks of the global system matrix. When implemented in parallel data locality becomes very important, as good data locality reduces the amount of communication required. Since data communication is usually the largest bottleneck of parallel code, we can expect DG code to perform well because of this.

The DG method is, through a suitable choice of numerical flux, a conservative method. For a physical problem where a quantity such as mass, energy or momentum is involved, conservation is an important property. Using a method that doesn't need adjustments to make it conservative, like DG, is another advantage.

The block structure means that the number of floating point operations (FLOPs) required to apply a DG mass matrix is proportional[1] to $(k+1)^{2d}$. The number of memory references required for the same operation is proportional to $(k+1)^d$. Here k is the polynomial degree used to approximate the solution in each cell and d is the geometric dimension of the domain of the PDE. So the number of FLOPs per memory reference is proportional to $(k+1)^d$ and for a fixed problem, k can be chosen high enough to attain a desired arithmetic intensity. This is an advantage of using a *higher order* DG method.

3.1 Overview

We start by looking at a PDE, describing transport, written as a conservation law:

$$\frac{\partial \boldsymbol{q}}{\partial t} + \nabla \cdot F(\boldsymbol{q}) = \boldsymbol{S}(\boldsymbol{q}) \tag{3.1}$$

[1] This can be reduced to $(k+1)^{d+1}$ with sum factorisation

We will solve this on some spatial domain, $\Omega \subset \mathbb{R}^d$, with appropriate boundary conditions. This equation governs the behaviour of the quantity of interest $\boldsymbol{q}(x,t)$ which is a vector valued, time dependent function $\boldsymbol{q} : \Omega \times [0,T] \to \mathbb{R}^m$. $F(\boldsymbol{q})$ represents some flux $F : \mathbb{R}^m \to \mathbb{R}^{m\times d}$, which we take the (column-wise) divergence of. That is $\nabla\cdot : \mathbb{R}^{m\times d} \to \mathbb{R}^m$, see equation (2.3) for the definition. $\boldsymbol{S}(\boldsymbol{q})$ represents some source term $\boldsymbol{S} : \mathbb{R}^m \to \mathbb{R}^d$. To solve the problem numerically we want to write this as a weak form and approximate $\boldsymbol{q}$ in a finite dimensional function space.

To start we will simplify the problem by considering a scalar quantity of interest $\phi : \Omega \times [0,T] \to \mathbb{R}$, where $m = 1$ in the discussion above we set. This leaves us with the general scalar conservation law:

$$\frac{\partial \phi}{\partial t} + \nabla \cdot \boldsymbol{F}(\phi) = S(\phi) \tag{3.2}$$

We will return to the more general vector conservation law in section 3.3.

First we must define a mesh. Since the DG method can use any shape of cell and hence work over arbitrary geometries, we consider a general mesh. It suffices to define the mesh $\mathcal{T}_h$ as a collection of cells $K \in \mathcal{T}_h$, which cover the entire space Ω and do not overlap. The subscript h denotes the maximal diameter of any cell.

We can define a discontinuous function space on this mesh as

$$\Phi_h = \{\phi_h \in L_2(\Omega) : \phi_h|_K \in P^k(K) \quad \forall K \in \mathcal{T}_h\}. \tag{3.3}$$

Where $P^k(K)$ denotes the space of polynomial functions of degree k defined on a cell K . To approximate ϕ in equation (3.2) we shall construct a $\phi_h \in \Phi_h$.

To define ϕ_h we can multiply our conservation law, equation (3.2), by a test function $\psi \in \Phi_h$ and integrate over a single cell

$$\int_K \frac{\partial \phi_h}{\partial t}\psi + (\nabla \cdot \boldsymbol{F}(\phi_h))\psi \; dx = \int_K S(\phi_h)\psi \; dx.$$

Later we will sum these cell contributions to obtain an integral over the whole domain Ω. To find an approximate solution $\phi_h \in \Phi_h$, we require this equation to hold for all $\psi \in \Phi_h$ and for all cells $K \in \mathcal{T}_h$.

We can integrate the flux term by parts, giving

$$\int_K \frac{\partial \phi_h}{\partial t}\psi \; dx + \int_{\partial K} \boldsymbol{F}(\phi_h) \cdot \boldsymbol{n}\psi \; ds - \int_K \boldsymbol{F}(\phi_h) \cdot \nabla\psi \; dx = \int_K S(\phi_h)\psi \; dx.$$

Note 3.1.1. The integral on the boundary ∂K is not well defined, since the solution is multi-valued on the facet between two cells. We introduce the numerical flux to address this, which will couple a cell to its neighbours. This is denoted by the function $\boldsymbol{F}^*(\phi_h^+, \phi_h^-)$, which takes as arguments the value of the solution on each side of the facet, distinguised with the labels $+$ and $-$.

Summing over all of the cells in our mesh, which cover the whole of the domain Ω, we

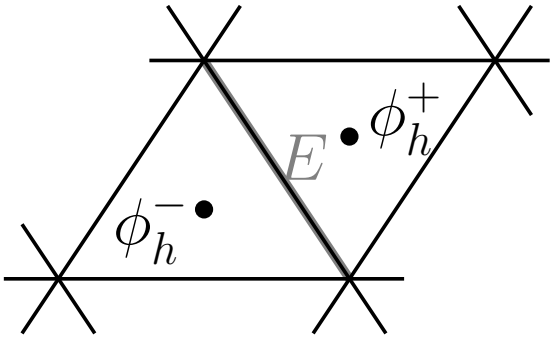

Figure 3-1: Sketch of two adjacent cells with plus and minus labelling to distinguish them over their shared facet

get:

$$\sum_{K\in\mathcal{T}_h}\left(\int_K \frac{\partial\phi_h}{\partial t}\psi\ dx + \int_{\partial K}\boldsymbol{F}^*(\phi_h^+,\phi_h^-)\cdot\boldsymbol{n}\psi\ ds - \int_K \boldsymbol{F}(\phi_h)\cdot\nabla\psi\ dx\right) = \sum_{K\in\mathcal{T}_h}\left(\int_K S(\phi_h)\psi\ dx\right) \tag{3.4}$$

The numerical flux is defined on each facet of the mesh. For any facet E, the values of ϕ_S^+ and ϕ_h^- are taken from the adjoining cells, shown in figure 3-1. This is the only part of the above equation that couples adjacent cells, hence the choice of this numerical flux is very important. We will discuss a specific choice of flux in our first example in section 3.2.

In order to write the equation in matrix form we write the approximate solution ϕ_h as a linear combination of basis functions. To do so we observe that there exists some finite set of basis functions $\{e_i \in \Phi_h\}$ and we can write the solution ϕ_h as

$$\phi_h(\boldsymbol{x},t) = \sum_i e_i(\boldsymbol{x})\phi_i(t). \tag{3.5}$$

This is the global solution, but it should be noted that the basis functions e_i have compact support on each cell and we will often consider the integral over a single cell without denoting ϕ_h or e_i differently. The approximation of ϕ on a single cell K is given by a function in $P^k(K)$, (the function space consisting of polynomials of degree k). In general the number of basis functions depends on the degree of the polynomial approximation k, the dimension of the domain d, and the number and shape of the cells in the mesh. An additional section discussing bases is included in appendix A.2.

To construct a matrix to represent the cell local mass bilinear form

$$\mathcal{M}_K(\phi_h,\psi) := \int_K \phi_h\psi\ dx, \qquad \text{for } \psi\in\Phi_h, \tag{3.6}$$

it is sufficient to consider one basis function of Φ_h at a time. The mass bilinear form on a single cell K is

$$\mathcal{M}_K(\phi_h,e_i) = \int_K \phi_h e_i\ dx = \int_K \left(\sum_j \phi_j e_j\right) e_i\ dx = \sum_j \left[\int_K e_j e_i\ dx\right]\psi_j$$

By defining the cell local mass matrix M_K element-wise as

$$M_{K,ij} := \int_K e_j e_i \, dx$$

the action on the *local* degrees of freedom vector, which is defined using equation (3.5)

$$\underline{\phi}_K := (\phi_1, \dots, \phi_j, \dots)^\top$$

is just $M_K \underline{\phi}_K$. That is $[M_K \underline{\phi}_K]_i = \mathcal{M}_K(\phi_h, e_i)$.

The *global* mass bilinear form is just the sum, over all of the cells in the mesh, of all of the local mass bilinear forms

$$\mathcal{M}(\phi_h, \psi) = \sum_{K \in \mathcal{T}_h} \mathcal{M}_K(\phi_h, \psi). \tag{3.7}$$

Since each local mass matrix M_K *only* acts on local degrees of freedom, we can write the *global* mass matrix as the direct sum of the local mass matrices

$$M = \bigoplus_{K \in \mathcal{T}_h} M_K.$$

Since this is the direct sum, the matrix M is block diagonal. Stacking all of the local DOF vectors for each cell on top of each other, that is to say

$$\underline{\phi} = (\underline{\phi}_{K_1}^\top, \dots, \underline{\phi}_{K_\ell}^\top, \dots)^\top,$$

we can represent the mass operator $\mathcal{M}(\partial_t \phi_h, \psi)$ as the matrix vector product $M(\partial_t \underline{\psi})$.

For equation (3.4), we can also define a bilinear form $\mathcal{A}$ using a similar technique, starting from the cell local bilinear form

$$\mathcal{A}_K(\phi_h, \psi) := \int_{\partial K} \boldsymbol{F}(\phi_h^+, \psi_h^-) \cdot \boldsymbol{n} \psi \, ds - \int_K \boldsymbol{F}(\phi_h) \cdot \nabla \psi + S(\phi_h)\psi \, dx.$$

Since $\mathcal{A}_K$ does *not* just use local degrees of freedom, now we have to consider contributions from neighbouring cells. If we assume that $\boldsymbol{F}$ is linear then we can represent $\mathcal{A}$ as a matrix, but this matrix is not block diagonal. This is because each neighbouring cell will contribute to an off diagonal block in the global matrix.

We set $\mathcal{A} = \sum_{K \in \mathcal{T}_h} \mathcal{A}_K$. Rather than carefully deriving the structure of A, the matrix approximating the bilinear form $\mathcal{A}$, for equation (3.1), we will do this explicitly in section 3.2 for an advection problem.

Using the operators we have defined, we can write the DG discretisation of equation (3.2) as

$$\mathcal{M}(\partial_t \phi_h, \psi) + \mathcal{A}(\phi_h, \psi) = 0 \qquad \forall \psi \in \Phi_h. \tag{3.8}$$

This is an abstract way of writing equation (3.4).

To demonstrate both implicit and explicit timestepping schemes, we will use the theta timestepping method to discretise this problem in time. The theta method for a general time dependent ordinary differential equation is:

$$\frac{d\phi}{dt} = f(t, \phi) \to \frac{\phi^{(n+1)} - \phi^{(n)}}{\Delta t} = \theta f\left(t^{(n+1)}, \phi^{(n+1)}\right) + (1-\theta) f\left(t^{(n)}, \phi^{(n)}\right)$$

For some known function f, suitable timestep Δt, with $t^{(n)} = t^{(0)} + n\Delta t$ and $\phi_h^{(n)}$ denotes $\phi_h\left(t^{(n)}\right)$.

If we do the same for equation (3.8), we obtain

$$\int_K \left(\frac{\phi_h^{(n+1)} - \phi_h^{(n)}}{\Delta t}\right) \psi \; dx = -\theta\mathcal{A}(\phi_h^{(n+1)}, \psi) - (1-\theta)\mathcal{A}(\phi_h^{(n)}, \psi)$$

For the theta method we can write terms at the next time step on the left and the current timestep on the right:

$$\mathcal{M}(\phi_h^{(n+1)}, \psi) + \Delta t\theta\mathcal{A}(\phi_h^{(n+1)}, \psi) = \mathcal{M}(\phi_h^{(n)}, \psi) - \Delta t(1-\theta)\mathcal{A}(\phi_h^{(n)}, \psi)$$

If $\boldsymbol{F}$ is linear, this can also be written as a matrix equation[2], with $\underline{\phi}^{(n+1)}$ representing the DOF vector for the solution $\phi_h(\boldsymbol{x}, t + n\Delta t)$ to be computed and $\underline{\phi}^{(n)}$ representing the DOF vector for the solution at the current time step:

$$(M + \Delta t\ \theta A)\underline{\phi}^{(n+1)} = (M - \Delta t(1-\theta))\underline{\phi}^{(n)}$$

Recall that $\mathcal{M}$ is represented by a block diagonal matrix M, but $\mathcal{A}$ represented by A has off block diagonal entries. If we consider an explicit timestepping scheme, such as explicit Euler, by setting $\theta = 0$, we only need to invert M at each time step. Since M is block diagonal, this can be done block-wise and in parallel, making explicit timestepping for a DG method *very* efficient.

One of the fundamental problems considered in this thesis is the much more difficult problem of inverting $(M + \Delta t\theta A)$ and how to efficiently use semi-implicit timestepping in conjunction with a DG method.

3.2 Advection

It is useful to motivate the general case in equation (3.1) by studying an example. Before we try to write down a DG discretisation of the shallow water equations, we consider a simpler problem, the linear advection-reaction equation. This is the method studied by Johnson and Pitkäranta and their paper [33] proves that for suitably smooth solutions to the advection equation the order of convergence for a DG method of degree p is $O(h^{p+1/2})$. We reproduce their method here to provide the details of how to construct a DG method for an advection problem and to highlight how upwinding is used as a numerical flux.

The advection-reaction equation is

$$\partial_t\phi + \nabla\cdot(\boldsymbol{\beta}\phi) + \alpha\phi = f \text{ on } \Omega \times [0, T]. \tag{3.9}$$

Equation (3.9) describes the movement of a scalar quantity of interest ϕ in some domain Ω. For this concrete example we consider a domain which is a unit square $\Omega = [0,1] \times [0,1]$. We can think of this equation modelling clouds being blown by the wind, or some contaminant flowing in a river. This equation does *not* model the diffusion of the material.

The function $\boldsymbol{\beta}(\boldsymbol{x}, t)$ is the advection vector, which determines the flow at every point in the domain. For this example we will only consider divergence free vector fields, which means that $\nabla\cdot\boldsymbol{\beta} = 0$, which implies that $\boldsymbol{\beta}\cdot\nabla\phi = \nabla\cdot(\boldsymbol{\beta}\phi)$. For now we just take $\boldsymbol{\beta}$ to be a constant vector over the domain, so all material flows in the same direction.

[2] If $\boldsymbol{F}$ is non-linear, we can replace $A\underline{\phi}$ in what follows with a non-linear function of the DOFs, $A(\underline{\phi})$

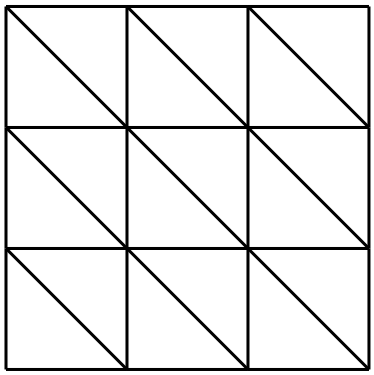

Figure 3-2: An example of a regular triangular mesh on a 2D square domain

The term $\alpha\phi$ is a reaction term, which changes the quantity of interest ϕ *depending* on the value of ϕ at any given point. In contrast f is some external forcing, which changes the quantity of interest independent of its value. If we take $\alpha = 0$ and $f = 0$ then no material is created or destroyed, and we expect the total amount of material in our domain to remain constant (unless it moves out of the domain).

Boundary conditions for equation (3.9) can be defined as follows: We will allow material to flow into and out of the domain, so the boundary is not a barrier. By doing so we only need to prescribe boundary conditions on the inflow boundary, that is all the points on $\partial\Omega$ where the advection vector $\boldsymbol{\beta}$ points into the domain. The inflow boundary is defined as $\Gamma_- := \{\boldsymbol{x} \in \partial\Omega : \boldsymbol{\beta}(\boldsymbol{x}) \cdot \boldsymbol{n} < 0\}$, where $\boldsymbol{n}$ is an outward pointing unit normal to Ω at the point $\boldsymbol{x}$. We can likewise define the outflow boundary as $\Gamma_+ := \{\boldsymbol{x} \in \partial\Omega : \boldsymbol{\beta}(\boldsymbol{x}) \cdot \boldsymbol{n} \geq 0\}$. Then our boundary condition is

$$\phi(\boldsymbol{x}) = g(\boldsymbol{x}, t) \qquad \forall \boldsymbol{x} \in \Gamma_-,$$

where g is a function describing the quantity of material that enters the domain Ω at each point on the inflow boundary. If $g = 0$, then no material enters Ω and material can only move out of the domain.

To formulate a DG discretisation, first we must define a mesh. For this 2D example we will use a regular triangular mesh $\mathcal{T}_h$, an example of which is shown in figure 3-2, containing cells $K \in \mathcal{T}_h$. Having a mesh allows us to construct a function space which is discontinuous on the cell boundaries:

$$\Phi_h := \{\phi \in L_2(\Omega) : \phi|_K \in P^k(K) \quad \forall K \in \mathcal{T}_h\}$$

Now we want to approximate the true solution ϕ by a function ϕ_h that lives in this discontinuous function space.

Once we have a mesh and a function space, we can multiply equation (3.9) by a test function $\psi \in \Phi_h$ and integrate over a single cell K

$$\int_K \partial_t \phi_h \psi + \boldsymbol{\beta} \cdot \nabla \phi_h \psi + \alpha \phi_h \psi \; dx = \int_K f\psi \; dx.$$

To solve equation (3.9), we must find $\phi_h \in \Phi_h$, such that the above equation holds for all $\psi \in \Phi_h$, and for all cells $K \in \mathcal{T}_h$.

We can integrate by parts to obtain

$$\int_K \partial_t \phi_h \psi - (\boldsymbol{\beta}\phi_h) \cdot \nabla\psi + \alpha\phi_h\psi \; dx + \int_{\partial K} (\boldsymbol{\beta}\phi_h)^* \cdot \boldsymbol{n} \; \psi \; ds = \int_K f\psi \; dx. \tag{3.10}$$

Where we have replaced the term $\boldsymbol{\beta}\phi_h$ in the boundary integral[3] with a numerical flux (still to be defined). We take the numerical flux to be the upwind flux, where the value of the flux *only* depends on the value of the solution on the upwind side. The choice of flux is important and we carefully describe its formulation in the next section.

Note that so far equation (3.10) only covers one cell, to obtain the global solution we must sum over all of the cells $K \in \mathcal{T}_h$. For the volume integrals this is not an issue, since for any $q \in \Phi_h$ we have $\sum_{K\in\mathcal{T}_h} \int_K q \; dx = \int_\Omega q \; dx$. However, the cell boundary integrals pose an issue. If part of the cell boundary coincides with the boundary of the domain, then we just impose the boundary conditions. If the cell boundary is partly or entirely contained in the interior of the domain (an interior facet of the mesh) then we pick up two contributions to the summation, one from each cell either side of the facet.

To deal with these we start by defining the set that contains all of the interior facets $\Gamma_{\text{int}} := \left(\bigcup_{K\in\mathcal{T}_h} \partial K\right) \setminus \partial\Omega$. Then we split the contributions of the cell boundary integrals into three: The inflow boundary, the outflow boundary and interior facets.

$$\begin{aligned}\int_\Omega \partial_t\phi_h\psi - \boldsymbol{\beta}\cdot\nabla\phi_h\psi + \alpha\phi_h\psi \; dx \\ + \underbrace{\int_{\Gamma_-} (\boldsymbol{\beta} g)\cdot\boldsymbol{n} \; \psi \; ds}_{\text{inflow}} + \underbrace{\int_{\Gamma_+} (\boldsymbol{\beta}\phi_h)\cdot\boldsymbol{n} \; \psi \; ds}_{\text{outflow}} + \underbrace{\int_{\Gamma_{\text{int}}} (\boldsymbol{\beta}\phi_h)^* \cdot (\boldsymbol{n}^-\psi^- + \boldsymbol{n}^+\psi^+) \; ds}_{\text{interior facets}} \\ = \int_\Omega f\psi \; dx.\end{aligned} \tag{3.11}$$

For the inflow and outflow boundaries there is only one value for the unit outward facing normal to take. On the inflow boundary information must be taken from the boundary condition, and on the outflow boundary information must be taken from the cell.

On the interior facets the flux through the facet is unique. The left-hand portion of figure 3-3 demonstrates the + and − labelling for a highlighted facet E. Over a given facet, some material is lost from cell on the upwind side (in figure 3-3 the '−' side) and is gained by the cell on the downwind side ('+' in figure 3-3), thus conserving the total amount of material. ψ^+ is the test function and $\boldsymbol{n}^+$ is the outward pointing normal for the cell labelled K^+ and likewise for the cell labelled K^-.

3.2.1 Numerical Flux

For this advection example we will use an upwind flux. The upwind flux always uses the value of the solution from the cell in the upwind direction to calculate the flux value, that is the solution value in the cell that $-\boldsymbol{\beta}$ points into. This means that information about ϕ_h always travels in the same direction as $\boldsymbol{\beta}$. In the case that $\boldsymbol{\beta}$ is parallel to a facet E then $\boldsymbol{\beta}\cdot\boldsymbol{n} = 0$ and there is no flux across E.

On each facet E, we give the cells each side arbitrary labels + and −, and define the solution on each side of an interface as

$$\phi_h^\pm := \lim_{\varepsilon\to 0} \phi_h(\boldsymbol{x} - \varepsilon\boldsymbol{n}^\pm), \quad x \in E,$$

[3]Note that ϕ_h is not single valued on the cell boundary

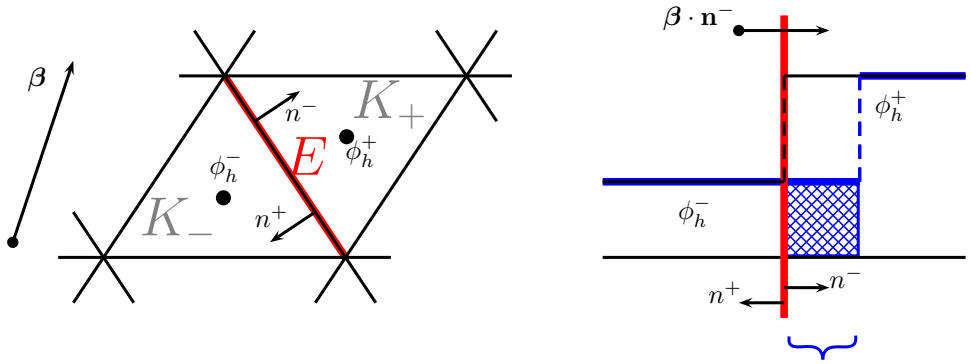

Figure 3-3: How the upwind flux is calculated through at the interface between two mesh cells

where $\boldsymbol{n}^{\pm}$ is the unit outward facing normal to the cell $K^{\pm}$.

Note the value of the solution at the interface, $\phi_h^{\pm}$, does not depend on $\boldsymbol{\beta}$. We do not know which side, $+$ or $-$, is the upwind side, but this can be found by taking the dot product of the cell normal with the advection vector. The numerical flux is defined as

$$(\boldsymbol{\beta}\phi_h)^* := \begin{cases} \boldsymbol{\beta}\phi_h^- & \text{if } n^-\cdot\boldsymbol{\beta} > 0 \\ \boldsymbol{\beta}\phi_h^+ & \text{if } n^-\cdot\boldsymbol{\beta} \leq 0. \end{cases} \tag{3.12}$$

This expression for the flux has the advantage of being symmetric in $+$ and $-$, that is if we replace all of the superscript $+$ with $-$ and vice versa, we get the same result.

Many authors in the DG literature use the "average" and "jump" operators, which are defined as follows:

$$\begin{aligned} \text{Average}: &\quad \{\{\phi\}\} := \frac{\phi^- + \phi^+}{2} \\ \text{(Scalar) Jump}: &\quad [\![\phi]\!] := \boldsymbol{n}^-\phi^- + \boldsymbol{n}^+\phi^+ \\ \text{(Vector) Jump}: &\quad [\![\boldsymbol{u}]\!] := \boldsymbol{n}^-\cdot\boldsymbol{u}^- + \boldsymbol{n}^+\cdot\boldsymbol{u}^+ \\ \text{(Matrix) Jump}: &\quad [\![M]\!] := M^-\boldsymbol{n}^- + M^+\boldsymbol{n}^+ \end{aligned}$$

Although these definitions may differ slightly from publication to publication, the ones above are commonly used, and are what we will use here. One advantage of using these operators, is they are symmetric in $+$ and $-$, so using only these operators in our definition of flux ensures its symmetry. As we refine our mesh and reduce the maximum cell diameter h, the jump terms will tend to 0 for sufficiently smooth ϕ. When it comes to analysing the DG method, by writing our flux using the jump notation it makes this fact easier to spot.

Note 3.2.1. One property of the jump operator is that, given a scalar argument, it returns a vector, and with a vector argument, it returns a scalar. For scalar quantities α, ϕ and vector quantities $\boldsymbol{\beta}, \boldsymbol{u}$ we have

- The value of $\alpha[\![\phi]\!]$ is a vector with the same number of components as $\boldsymbol{n}$,
- The value of $\alpha[\![\boldsymbol{u}]\!]$ is a scalar (but only defined if $\boldsymbol{u}$ and $\boldsymbol{n}$ are the same size),
- The value of $\boldsymbol{\beta}\cdot[\![\phi]\!]$ is a scalar (but only defined if $\boldsymbol{\beta}$ and $\boldsymbol{n}$ are the same size),
- The value of $\boldsymbol{\beta}[\![\boldsymbol{u}]\!]$ is a vector that has the same number of components as $\boldsymbol{\beta}$ (but only defined if $\boldsymbol{u}$ and $\boldsymbol{n}$ are the same size).

Using these operators we can write the upwind flux as

$$(\boldsymbol{\beta}\phi_h)^* := \{\{\boldsymbol{\beta}\phi_h\}\} + \frac{1}{2}\,|\boldsymbol{\beta}\cdot\boldsymbol{n}|\,[\![\phi_h]\!], \tag{3.13}$$

where we have not needed to specify which normal we use, since $|\boldsymbol{\beta}\cdot\boldsymbol{n}^+| = |\boldsymbol{\beta}\cdot\boldsymbol{n}^-|$. This form of the numerical flux is convenient as it avoids having to evaluate any "if" condition.

We carefully derive the upwind flux in terms of the jump and average operators in appendix A.1. The equivalence makes use of the vector

$$\boldsymbol{v}^{\pm} = \frac{1}{2}(\boldsymbol{\beta} + |\boldsymbol{\beta}\cdot\boldsymbol{n}|\cdot\boldsymbol{n}^{\pm}), \tag{3.14}$$

which is later used in equation (3.20). This gives the value of the flux on the upwind side when the normal and advection vector point in the same direction, but zero otherwise.

By writing the flux in this form it allows us to write the global problem, equation (3.11), as

$$\begin{aligned}\int_\Omega \partial_t\phi_h\psi - \boldsymbol{\beta}\cdot\nabla\phi_h\psi + \alpha\phi_h\psi\;dx \\ + \int_{\Gamma_-} (\boldsymbol{\beta}g)\cdot\boldsymbol{n}\;\psi\;ds + \int_{\Gamma_+} (\boldsymbol{\beta}\phi_h)\cdot\boldsymbol{n}\;\psi\;ds \\ + \int_{\Gamma_{\text{int}}} \left(\{\{\boldsymbol{\beta}\phi_h\}\} + \frac{1}{2}\,|\boldsymbol{\beta}\cdot\boldsymbol{n}|\,[\![\phi_h]\!]\right)\cdot[\![\psi]\!]\;ds \\ = \int_\Omega f\psi\;dx.\end{aligned}$$

3.2.2 Matrices

To turn our weak form into a matrix equation, we expand our approximate solution ϕ_h in terms of its basis functions, e_j:

$$\phi_h(\boldsymbol{x},t) = \sum_j \phi_j(t)e_j(\boldsymbol{x}), \quad \text{where} \quad \phi_j(t) := \phi_h(\boldsymbol{x}_j,t) \tag{3.15}$$

Where e_j forms a nodal basis.

To talk precisely about the basis for any finite element, we use the Ciarlet definition of a finite element [16, 39]. A finite element is defined to be the triple $(K,\mathcal{V},\mathcal{L})$, where K is the domain (bounded, closed subset of $\mathbb{R}^d$, non-empty interior, piecewise smooth boundary), $\mathcal{V}$ is a finite dimensional function space on K with dimension n and $\mathcal{L} = \{\ell_0,\ldots,\ell_{n-1}\}$ a basis of the dual space $\mathcal{V}'$ (a set of bounded linear functionals on $\mathcal{V}$), referred to as the set of degrees of freedom, or nodes.

With this definition of a finite element it is possible to define a basis for $\mathcal{V}$, which is dual to $\mathcal{L}$. This basis $\{e_0,\ldots,e_{n-1}\}$ has the property that $\ell_i(e_j) = \delta_{ij}$. The set $\{e_0,\ldots,e_{n-1}\}$ is called a nodal basis and we only consider nodal bases in this thesis.

The $\boldsymbol{x}_j$ in equation (3.15) are points in the domain K. One of the simplest set of degrees freedom (choice fo $\mathcal{L}$) is just point evaluation, that is $\ell_i(e) = e(\boldsymbol{x}_i)$, which we use here to obtain our approximate solution.

We can collect together all of the nodal bases $\{e_0,\ldots,e_{n-1}\}$ for all of the domains, which in our case correspond to the cells K in our mesh $\mathcal{T}_h$ to form a basis for the whole domain Ω. We implicitly do this for the rest of the thesis to avoid having to include a local DOF to global DOF map in all of our terms.

For now we do not fix a basis, and just state that we sum over all the basis functions (covering the whole domain Ω), indexed here by j. A discussion of the choice of basis functions is included in appendix A.2.

To build the matrices for the local problem, we write the global system in terms of a summation over cells and edges

$$\begin{aligned}\sum_{K\in\mathcal{T}_h}\int_K \partial_t\phi_h\psi\ dx = \sum_{K\in\mathcal{T}_h}\int_K \boldsymbol{\beta}\cdot\nabla\phi_h\psi - \alpha\phi_h\psi\ dx \\ - \sum_{E\in\mathcal{E}(\Gamma_-)}\int_E(\boldsymbol{\beta}g)\cdot\boldsymbol{n}\ \psi\ ds - \sum_{E\in\mathcal{E}(\Gamma_+)}\int_E(\boldsymbol{\beta}\phi_h)\cdot\boldsymbol{n}\ \psi\ ds \\ - \sum_{E\in\mathcal{E}(\Gamma_{\text{int}})}\int_E\left(\{\{\boldsymbol{\beta}\phi_h\}\} + \frac{1}{2}\ |\boldsymbol{\beta}\cdot\boldsymbol{n}|\ [\![\phi_h]\!]\right)\cdot[\![\psi]\!]\ ds \\ + \sum_{K\in\mathcal{T}_h}\int_K f\psi\ dx, \end{aligned} \tag{3.16}$$

where the notation $E\in\mathcal{E}(\Gamma)$ is shorthand for: Take E, a straight edge or facet that forms the interface between two cells, or one cell and the boundary, from the collection $\mathcal{E}(\Gamma)$ of facets that make up the set Γ.

We write equation (3.16) abstractly in terms of the bilinear forms $\mathcal{M}$ and $\mathcal{L}$, as well as a linear operator $\mathcal{S}$ that includes all of the source and boundary terms

$$\mathcal{M}(\partial_t\phi_h,\psi) = \mathcal{L}(\phi_h,\psi) + \mathcal{S}(\psi), \tag{3.17}$$

where $\mathcal{M}$ was defined in equation (3.7),

$$\begin{aligned}\mathcal{L}(\phi_h,\psi) := \sum_{K\in\mathcal{T}_h}\int_K \boldsymbol{\beta}\cdot\nabla\phi_h\psi - \alpha\phi_h\psi\ dx - \sum_{E\in\mathcal{E}(\Gamma_+)}\int_E(\boldsymbol{\beta}\phi_h)\cdot\boldsymbol{n}\ \psi\ ds \\ - \sum_{E\in\mathcal{E}(\Gamma_{\text{int}})}\int_E\left(\{\{\boldsymbol{\beta}\phi_h\}\} + \frac{1}{2}\ |\boldsymbol{\beta}\cdot\boldsymbol{n}|\ [\![\phi_h]\!]\right)\cdot[\![\psi]\!]\ ds \end{aligned} \tag{3.18}$$

and

$$\mathcal{S}(\psi) := -\sum_{E\in\mathcal{E}(\Gamma_-)}\int_E(\boldsymbol{\beta}g)\cdot\boldsymbol{n}\ \psi\ ds + \sum_{K\in\mathcal{T}_h}\int_K f\psi\ dx \tag{3.19}$$

To obtain the mass matrix M from the bilinear form $\mathcal{M}$ above, refer back to the discussion in section 3.1, starting from equation (3.6).

Constructing the matrix L for the bilinear form $\mathcal{L}$ is much more involved, due to the number of terms and the fact it includes the coupling between cells. We will work through the right hand side of equation (3.16) term by term and then indicate diagrammatically in figure 3-5 where in the global system matrix each term has contributed. The vector $\underline{S}$ representing the linear functional $\mathcal{S}$ is also shown in figure 3-5.

First the cell "volume" term, which is indicated in figure 3-4 by the blue triangles and is the first term of equation (3.18).

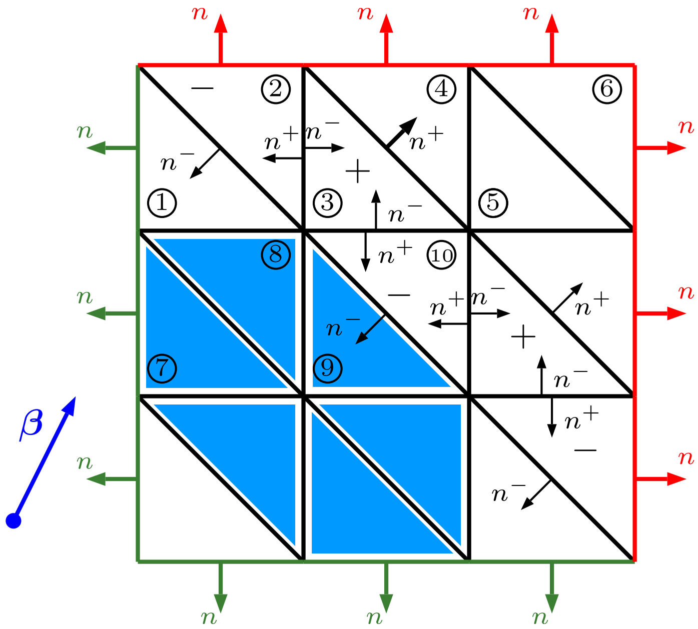

Figure 3-4: Mesh labelled to indicate the highlighted "volume", inflow boundary, outflow boundary and facet terms in the global weak form:

$$\sum_{K\in\mathcal{T}_h}\int_K \partial_t\phi_h\psi\ dx = \sum_{K\in\mathcal{T}_h}\int_K \boldsymbol{\beta}\cdot\nabla\phi_h\psi - \alpha\phi_h\psi\ dx$$
$$- \sum_{E\in\mathcal{E}(\Gamma_-)}\int_E (\boldsymbol{\beta}g)\cdot\boldsymbol{n}\ \psi\ ds - \sum_{E\in\mathcal{E}(\Gamma_+)}\int_E (\boldsymbol{\beta}\phi_h)\cdot\boldsymbol{n}\ \psi\ ds$$
$$- \sum_{E\in\mathcal{E}(\Gamma_{\text{int}})}\underline{\int_E \left(\{\{\boldsymbol{\beta}\phi_h\}\} + \frac{1}{2}\ |\boldsymbol{\beta}\cdot\boldsymbol{n}|\ [\![\phi_h]\!]\right)\cdot[\![\psi]\!]\ ds}$$
$$+ \sum_{K\in\mathcal{T}_h}\int_K f\psi\ dx$$

We can write the integral in the first term of equation (3.18) as

$$\int_K \boldsymbol{\beta}\cdot\nabla\phi_h\psi - \alpha\phi_h e_i \; dx = \int_K \sum_j (\boldsymbol{\beta}\cdot\nabla e_j)\phi_j e_i - \sum_j \alpha\phi_j e_j e_i \; dx$$
$$= \sum_j \left[\int_K (\boldsymbol{\beta}\cdot\nabla e_j - \alpha e_j)e_i \; dx\right]\phi_j.$$

The term in square brackets gives the (element–wise) contributions to the local matrix. Note that, since the volume term does not contain any coupling, it only contributes to the block diagonal of the matrix L, just like in the mass matrix. This is indicated by all of the matrix blocks (black squares) containing a blue spot in figure 3-5.

The integral over the inflow boundary, the green edges in figure 3-4 and first term of equation (3.19), remains the same. Since this term does not depend on the solution vector $\underline{\phi}$, this term does not have to form part of the global matrix, but can be included as an additional vector, $\underline{S}$. These are indicated in figure 3-5 by rectangles containing green spots.

The integral over the outflow boundary, the red edges in figure 3-4 and the second term of equation (3.18), can be written

$$\int_E (\boldsymbol{\beta}\phi_h)\cdot\boldsymbol{n}\; e_i \; ds = \int_E \sum_j (\boldsymbol{\beta}\cdot\boldsymbol{n})\phi_j e_j e_i \; ds$$
$$= \sum_j \left[\int_E (\boldsymbol{\beta}\cdot\boldsymbol{n})e_j e_i \; ds\right]\phi_j.$$

Unlike the inflow boundary this term depends on the solution $\underline{\phi}$, so does get included in

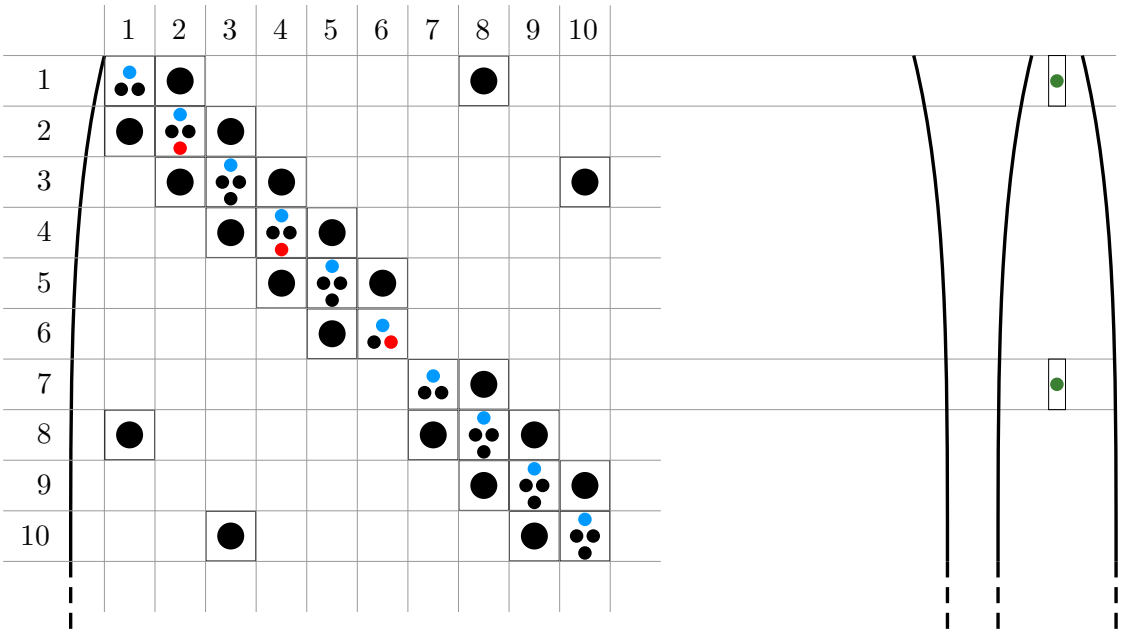

Figure 3-5: A representation of the matrix L and vector $\underline{S}$ indicating the location of the terms that appear in equation (3.16) as indicated in figure 3-4. Black squares containing spots indicate block matrices and rectangles indicate block vectors. The colour of the spot indicate which terms of the equation contribute to that block.

the global matrix. This only contributes to certain block diagonals, indicated in figure 3-5 by black squares containing red spots.

Integrals over internal facets, black edges in figure 3-4 and the third term of equation (3.18), also contribute to the global matrix, each facet integral contributes to four matrix blocks. For brevity we use the vectors $\boldsymbol{v}^{\pm}$ that we defined in equation (3.14) and their point-wise evaluation, $\boldsymbol{v}_j^{\pm}$.

$$\begin{aligned}
&\int_E \left(\{\{\boldsymbol{\beta}\phi_h\}\} + \frac{1}{2}\,|\boldsymbol{\beta}\cdot\boldsymbol{n}|\,[\![\phi_h]\!] \right) \cdot [\![e_i]\!]\; ds \\
&\qquad = \quad \int_E (\boldsymbol{v}^+\cdot\boldsymbol{n}^+)\phi_h^+ e_i^+\; ds + \int_E (\boldsymbol{v}^-\cdot\boldsymbol{n}^+)\phi_h^- e_i^+\; ds \\
&\qquad\quad + \int_E (\boldsymbol{v}^+\cdot\boldsymbol{n}^-)\phi_h^+ e_i^-\; ds + \int_E (\boldsymbol{v}^-\cdot\boldsymbol{n}^-)\phi_h^- e_i^-\; ds \\
&\qquad = \quad \int_E \sum_j (\boldsymbol{v}_j^+\cdot\boldsymbol{n}^+)\phi_j^+ e_j^+ e_i^+\; ds + \int_E \sum_j (\boldsymbol{v}_j^-\cdot\boldsymbol{n}^+)\phi_j^- e_j^- e_i^+\; ds \\
&\qquad\quad + \int_E \sum_j (\boldsymbol{v}_j^+\cdot\boldsymbol{n}^-)\phi_j^+ e_j^+ e_i^-\; ds + \int_E \sum_j (\boldsymbol{v}_j^-\cdot\boldsymbol{n}^-)\phi_j^- e_j^- e_i^-\; ds \\
&\qquad = \quad \sum_j \left[\int_E (\boldsymbol{v}_j^+\cdot\boldsymbol{n}^+) e_j^+ e_i^+\; ds \right] \phi_j^+ + \sum_j \left[\int_E (\boldsymbol{v}_j^-\cdot\boldsymbol{n}^+) e_j^- e_i^+\; ds \right] \phi_j^- \\
&\qquad\quad + \sum_j \left[\int_E (\boldsymbol{v}_j^+\cdot\boldsymbol{n}^-) e_j^+ e_i^-\; ds \right] \phi_j^+ + \sum_j \left[\int_E (\boldsymbol{v}_j^-\cdot\boldsymbol{n}^-) e_j^- e_i^-\; ds \right] \phi_j^- \qquad (3.20)
\end{aligned}$$

The terms in square brackets in equation (3.20) form the entries of four matrix blocks. Each facet term contributes these four matrices to the global matrix. This can be seen more readily by explicitly defining

$$Q_{++} := \int_E (\boldsymbol{v}_j^+\cdot\boldsymbol{n}^+) e_j^+ e_i^+\; ds,$$

and likewise for Q_{-+}, Q_{+-}, Q_{--}, in the order above. Then collecting the degrees of freedom for each of the cells K_+ and K_- into the vector

$$\underline{\phi}_{K\pm} = (\phi_1^{\pm}, \ldots, \phi_j^{\pm}, \ldots)^{\top},$$

and stacking these two vectors on top of each other

$$\underline{\widetilde{\phi}} = \begin{pmatrix} \underline{\phi}_{K+} \\ \underline{\phi}_{K-} \end{pmatrix},$$

we see that the four block matrices obtained from a equation (3.20) can be writen as

$$\begin{pmatrix} Q_{++} & Q_{-+} \\ Q_{+-} & Q_{--} \end{pmatrix} \begin{pmatrix} \underline{\phi}_{K+} \\ \underline{\phi}_{K-} \end{pmatrix}.$$

The location of these matrix blocks are indicated in figure 3-5 by squares containing black spots. There are many such blocks in the diagram, some with more than one black

spot in. To see why, note that the edge between cells 1 and 2 contributes entries in the $(1,1),(1,2),(2,1)$ and $(2,2)$ blocks and the edge between cells 2 and 3 entries in the $(2,2),(2,3),(3,2)$ and $(3,3)$ blocks, and so on for all of the internal facets.

Finally, the source term, which is the second term of equation (3.19), can be written, again using a point-wise approximation of f

$$\int_E f e_i \, ds = \int_E f_j e_i \, ds.$$

Like the inflow boundary term, this has no dependence on the solution ϕ_h, and can be put into the vector of source terms $\underline{S}$.

We do not write down the full matrix L – the discrete form of the operator $\mathcal{L}$ from equation (3.17). Not only would it be large and complicated, but it would not give any more insight than the diagram in figure 3-5. This discussion should be sufficient to convince the reader that one would not want to construct these DG matrices manually.

Instead we will use the software framework, Firedrake, to automatically assemble these matrices. This is covered in detail in chapter 7, where the implementation of the DG method using Firedrake is discussed. By using such a framework, we minimise the chances of incorrectly constructing the DG matrices and can focus on efficiently solving the linear systems of equations that result.

All of this means we can solve the PDE problem

$$\mathcal{M}(\partial_t \phi_h, \psi) = \mathcal{L}(\phi_h, \psi) + \mathcal{S}(\psi)$$

by solving the system of ODEs

$$M(\partial_t \underline{\phi}) = L\underline{\phi} + \underline{S}.$$

3.2.3 Timestepping

The semi-discretised problem we have to solve is

$$M(\partial_t \underline{\phi}) - L\underline{\phi} = \underline{S} \tag{3.21}$$

where M, L and $\underline{\phi}$ were defined in section 3.2.2. The term $\partial_t \underline{\phi}$ represents the time derivative of the (time-dependent) coefficients in $\underline{\phi}$.

To discretise in time we must approximate $\partial_t \underline{\phi}$. For a general time dependent problem $\partial_t \underline{\phi} = f(t, \underline{\phi})$, we can write a general explicit timestepping scheme as

$$\underline{\phi}^{(n+1)} = \underline{\phi}^{(n)} + (\Delta t) P(t^{(n)}, \underline{\phi}^{(n)}). \tag{3.22}$$

where P is some vector valued function that may explicitly depend on the current time $t^{(n)}$ and the vector $\underline{\phi}^{(n)}$ (the solution DOF vector evaluated at time $t^{(n)}$). As before $t^{(n)} = t + n\Delta t$.

This can be made more concrete by choosing some timestepping method. For explicit Euler method (seen in chapter 2) P is just

$$P(t, \underline{\phi}) = f(t, \underline{\phi})$$

or for our time dependent linear advection problem in equation (3.21), equation (3.22) becomes

$$\underline{\phi}^{(n+1)} = \underline{\phi}^{(n)} + (\Delta t) M^{-1}(L\underline{\phi}^{(n)} + \underline{S}(t^{(n)})).$$

We could use explicit Euler for timestepping, but an explicit timestepping method becomes unstable as the order of the DG method increases. That is, even if we reduce Δt so that the CFL condition (equation (2.14)) holds, explicit Euler will still be an unstable method. If Δt decreases at the rate $O(h^{(1+\varepsilon)})$, for some $\varepsilon > 0$, then we can regain stability [20] but this imposes greater restriction on the timestep size.

A better choice for an explicit timestepping scheme for a DG discretisation would be a strong stability preserving Runge-Kutta 3 stage method (SSP RK3) [49]. This method has better stability at higher polynomial degree approximations of the solution ϕ_h.

Note 3.2.2. Strong stability preserving methods are used for conservation laws to prevent the unphysical growth of solutions. If it is true that: When an explicit Euler timestepping scheme is used and $|c\Delta t/\Delta x| \leq 1$ is satisfied, then $\|\phi^{(n+1)}\| \leq \|\phi^{(n)}\|$, for some norm $\|\phi\|$ on the solution ϕ. Then it is possible to use a higher order scheme that preserves the strong stability condition $\|\phi^{(n+1)}\| \leq \|\phi^{(n)}\|$.

For this problem and for a time step Δt, define the vectors

$$\begin{aligned}
\underline{k}_1 &= f(t^{(n)}, \underline{\phi}^{(n)}) \\
&= \quad M^{-1}(L\underline{\phi}^{(n)} + \underline{S}(t^{(n)})) \\
\underline{k}_2 &= f\left(t^{(n)} + \Delta t,\ \underline{\phi}^{(n)} + \Delta t\underline{k}_1\right) \\
&= \quad M^{-1}(L(\underline{\phi}^{(n)} + \Delta t k_1) + \underline{S}(t^{(n)} + \Delta t)) \\
\underline{k}_3 &= f\left(t^{(n)} + \frac{\Delta t}{2},\ \underline{\phi}^{(n)} + \frac{\Delta t}{4}\underline{k}_1 + \frac{\Delta t}{4}\underline{k}_2\right) \\
&= \quad M^{-1}\left(L\left(\underline{\phi}^{(n)} + \frac{\Delta t}{4}\underline{k}_1 + \frac{\Delta t}{4}\underline{k}_2\right) + \underline{S}\left(t^{(n)} + \frac{\Delta t}{2}\right)\right),
\end{aligned}$$

so that we can write the explicit timestepping function P as[4]

$$P(t, \underline{\phi}^{(n)}) = \frac{\Delta t}{6}(\underline{k}_1 + \underline{k}_2 + 4\underline{k}_3).$$

The update step is

$$\underline{\phi}^{(n+1)} = \underline{\phi}^{(n)} + \frac{\Delta t}{6}(\underline{k}_1 + \underline{k}_2 + 4\underline{k}_3).$$

We must invert the mass matrix M three times each update step. However, M is block diagonal, so this can be done very efficiently. Furthermore, we can avoid storing the values of k_i in separate vectors by evaluating them one at a time and accumulating them in the update step. This will then reduce the amount of memory required, since the vector $\underline{\phi}$ will be large.

For any of the discretisations we discuss to be convergent we need to ensure two things:

- **Stability:** A single step method is stable if there exists some $\eta \in \mathbb{R}$ independent of

[4] This form of the SSP RK3 method may look unfamiliar as it is usually written in 3 stages:

$$\begin{aligned}
\underline{\phi}^{(1)} &= \underline{\phi}^{(n)} + \Delta t f(t^{(n)},\ \underline{\phi}^{(n)}) \\
\underline{\phi}^{(2)} &= \frac{3}{4}\underline{\phi}^{(1)} + \frac{1}{4}\left(t + \Delta t,\ \Delta t f(\underline{\phi}^{(1)})\right) \\
\underline{\phi}^{(n+1)} &= \frac{1}{3}\underline{\phi}^{(2)} + \frac{2}{3}\left(t + \frac{1}{2}\Delta t,\ \Delta t f(\underline{\phi}^{(2)})\right)
\end{aligned}$$

Both are equivalent, but we will refer to the form in the main text in chapter 4.

the discretisation, such that $\left|\frac{\phi^{(n+1)}}{\phi^{(n)}}\right| \leq 1 + \eta\Delta t$. There are alternative definitions of stability, which guaruntee more properties, such as A-stability and L-stability (see [37]).

- **Consistency:** A timestepper is consistent if the truncation error,

$$\varepsilon^{(n+1)} := \frac{1}{\Delta t}(\phi^{(n+1)} - \phi^{(n)}) - \frac{\partial \phi}{\partial t},$$

goes to zero $\varepsilon^{(n+1)} \to 0$ as $\Delta t \to 0$. A method is consistent of order p if the truncation error goes to zero at rate p.

This ensures that as we increase the resolution of the discretisation we will converge to the true solution. A timestepping method is convergent of order p if it is both stable and consistent of order p.

3.3 Shallow Water Equations

We provide a DG discretisation of the shallow water equations, which will be the base for the subsequent chapters. The convergence of a DG discretisation of the SWE is studied by Bui-Thanh [15](which also contains the theory for the hybridised case). The paper finds that for a suitably smooth solutions and small enough timestep, a convergence rate of $O(h^{p+1/2})$ can be obtained when using a degree p DG method. This is the same rate as Johnson and Pitkäranta find with the advection problem studies in section 3.2.

To perform a DG discretisation of the shallow water equations we require a vector conservation law, similar to equation (3.1)

$$\frac{\partial \boldsymbol{q}}{\partial t} + \nabla \cdot F(\boldsymbol{q}) = \boldsymbol{S}(\boldsymbol{q}). \tag{3.23}$$

We can define a discontinuous function space on a mesh $\mathcal{T}_h$ as

$$\mathcal{Q}_h = \{\boldsymbol{q}_h \in [L_2(\Omega)]^m : \boldsymbol{q}_h|_K \in [P^k(K)]^m \quad \forall K \in \mathcal{T}_h\}. \tag{3.24}$$

If we follow through the process in section 3.1, the time dependent weak form of equation (3.1) with vector valued functions is: Seek $\boldsymbol{q}_h \in \mathcal{Q}_h$ so that

$$\sum_{K\in\mathcal{T}_h}\left(\int_K \frac{\partial \boldsymbol{q}_h}{\partial t}\cdot\boldsymbol{p}\,dx + \int_{\partial K} F^*(\boldsymbol{q}_h^+,\boldsymbol{q}_h^-)\boldsymbol{p}\cdot\boldsymbol{n}\,ds - \int_K F(\boldsymbol{q}_h):\nabla\boldsymbol{p}\,dx\right) = \sum_{K\in\mathcal{T}_h}\left(\int_K \boldsymbol{S}(\boldsymbol{q}_h)\cdot\boldsymbol{p}\,dx\right)$$

for all $\boldsymbol{p} \in \mathcal{Q}_h$. We can define suitable bilinear forms to write this abstractly as: Find $\boldsymbol{q}_h \in \mathcal{Q}_h$ so that

$$\mathcal{M}(\partial_t \boldsymbol{q}_h, \boldsymbol{p}) + \mathcal{A}(\boldsymbol{q}_h, \boldsymbol{p}) = 0 \qquad \forall \boldsymbol{p} \in \mathcal{Q}_h \tag{3.25}$$

At which point we can use a timestepping method to advance the solution in time, just as we did in section 3.1.

In the context of numerical weather prediction, there are many desirable properties of the model that we would like in our discretisation. Staniforth and Thuburn [50] outline ten points, which we summarise here:

1. Mass conservation
2. Accurate representation of balanced flows (hydrostatic and geostrophic)
3. No computational modes, or at least controlled
4. Geopotential and pressure gradient shouldn't produce unphysical sources of vorticity
5. Terms involving pressure gradient should be energy conserving
6. Discretisation of Coriolis force should be energy conserving
7. Rossby waves shouldn't propagate unrealistically fast
8. Axial angular momentum should be conserved
9. Approximately second order accuracy
10. Minimal grid imprinting

Since we are only trying to perform shallow water simulations, not all points are relevant here.

The continuum equations for SWE are conservation laws and DG is a naturally conservative method, we conserve not only mass but also momentum. We demonstrate accurate representation of balanced flows using an analytic solution for a balanced vortex, discussed in section 2.3. Once computational mode is the zero frequency "slow" wave that does not propagate, but this is by choice of constant Coriolis, so is controlled.

Points 4-8 rely on the method being mimetic, that is the choice of finite elements mimic the continuum vector calculus identities. Since we do not use mimetic finite elements, these properties are not guaranteed, this includes no guarantee that potential vorticity is conserved. If we wanted to explicitly conserve potential vorticity, it is possible to reformulate the conservation law in terms of the potential vorticity and recover the height potential and momentum from the new system, but we do not do this.

We demonstrate excellent accuracy at different rates for different polynomial degrees in chapter 8. Finally grid imprinting is minimised by using small grid spacing, but this point is much less relevant as we only look at planar grids. This point is much more relevant for spherical grids, which contain "special points" around which the numerical solution may be notably different to the region around it.

3.3.1 Non-linear shallow water equations

The SWE describe the evolution of the height and momentum perturbation of a body of water by using a conservation of mass equation and conservation of momentum equation respectively. The properties of the SWE were outlined in chapter 2, in this section we will perform a DG discretisation of the equations.

The non-linear SWE in two spatial dimensions, defined on some region $\Omega \subset \mathbb{R}^2$, for some time interval $[0, T_0]$, can be written as

$$\frac{\partial \phi_S}{\partial t} + \nabla \cdot \boldsymbol{U} = 0 \tag{3.26}$$

$$\frac{\partial \boldsymbol{U}}{\partial t} + \nabla \cdot \left[\frac{\boldsymbol{U} \otimes \boldsymbol{U}}{\phi_B + \phi_S} + \left(\frac{1}{2}\phi_S^2 + \phi_B \phi_S \right) \mathcal{I}_2 \right] = -f(\hat{k} \times \boldsymbol{U}) + \phi_S \nabla \phi_B. \tag{3.27}$$

Equations (3.26) and (3.27) are the same as equations (2.1) and (2.2). For this discussion consider the square domain $\Omega = [0,1] \times [0,1]$ with periodic boundary conditions.

By defining the vector of quantities $\boldsymbol{q} = (\phi_S, \boldsymbol{U}^\top)^\top = (\phi_S, U_1, U_2)^\top$, we define the flux function $F : \mathbb{R}^3 \to \mathbb{R}^{3\times 2}$ as

$$\begin{aligned} F(\boldsymbol{q}) &= \begin{pmatrix} \boldsymbol{U}^\top \\ \frac{\boldsymbol{U}\otimes\boldsymbol{U}}{\phi_S+\phi_B} + \frac{1}{2}(\phi_S^2 + 2\phi_B\phi_S)\mathcal{I}_2 \end{pmatrix} \\ &= \begin{pmatrix} U_1 & U_2 \\ \frac{U_1^2}{\phi_S+\phi_B} + \frac{1}{2}(\phi_S^2 + 2\phi_B\phi_S) & \frac{U_1U_2}{\phi_S+\phi_B} \\ \frac{U_1U_2}{\phi_S+\phi_B} & \frac{U_2^2}{\phi_S+\phi_B} + \frac{1}{2}(\phi_S^2 + 2\phi_B\phi_S) \end{pmatrix} \end{aligned}$$

and source function $\boldsymbol{S} : \mathbb{R}^3 \to \mathbb{R}^3$ as

$$\begin{aligned} \boldsymbol{S}(\boldsymbol{q}) &= -\begin{pmatrix} 0 \\ f(\hat{k} \times \boldsymbol{U}) - \phi_S \nabla \phi_B \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ fU_2 + \phi_S(\partial_x \phi_B) \\ -fU_1 + \phi_S(\partial_y \phi_B) \end{pmatrix}. \end{aligned}$$

Using this formulation, the shallow water equations can be written as a conservation law we saw in equation (3.23).

To describe this, we start by defining a mesh $\mathcal{T}_h$. We will use the same mesh as in section 3.2 on our domain $\Omega = [0,1] \times [0,1]$ (T_h is a mesh of triangular cells with maximum diameter h).

We define two discontinuous function spaces, one for the scalar valued function ϕ_S and one for the vector valued function $\boldsymbol{U}$.

$$\begin{aligned} \Phi_h &= \{\phi_h \in L_2(\Omega) : \phi_h|_K \in P^k(K) \quad \forall K \in \mathcal{T}_h\} \\ \mathcal{U}_h &= \{\boldsymbol{U}_h \in [L_2(\Omega)]^2 : \boldsymbol{U}_h|_K \in [P^k(K)]^2 \quad \forall K \in \mathcal{T}_h\} \end{aligned} \tag{3.28}$$

Now we approximate $\boldsymbol{q}$ by $\boldsymbol{q}_h \in (\Phi_h \times \mathcal{U}_h)$ and multiply equation (3.23) by a test function $\boldsymbol{p} = (\psi, V_1, V_2)^\top$ and integrate over a single mesh cell K, to obtain the weak problem: Find $\boldsymbol{q}_h \in (\Phi_h \times \mathcal{U}_h)$, the discontinuous function space, such that

$$\int_K \left(\frac{\partial \boldsymbol{q}_h}{\partial t} + \nabla \cdot F(\boldsymbol{q}_h) \right) \cdot \boldsymbol{p} \; dx = \int_K \boldsymbol{S}(\boldsymbol{q}_h) \cdot \boldsymbol{p} \; dx, \tag{3.29}$$

for all $\boldsymbol{p} \in (\Phi_h \times \mathcal{U}_h)$, for all $K \in \mathcal{T}_h$.

Using Green's identity (integration by parts) on the divergence term we obtain

$$\int_K \frac{\partial \boldsymbol{q}_h}{\partial t} \cdot \boldsymbol{p} - F(\boldsymbol{q}_h) : \nabla \boldsymbol{p} \; dx + \int_{\partial K} [F^*(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+)\boldsymbol{n}] \cdot \boldsymbol{p} \; ds = \int_K \boldsymbol{S}(\boldsymbol{q}_h) \cdot \boldsymbol{p} \; dx. \qquad \forall K \in \mathcal{T}_h$$

Since the term on the left-hand side of equation (3.29) is not single valued, the function F has been replaced with the numerical flux $F^* : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^{3\times 2}$ taking values from both sides of the facet.

If we sum over all cells, to obtain a global problem we get

$$\int_\Omega \frac{\partial \boldsymbol{q}_h}{\partial t} \cdot \boldsymbol{p} - F(\boldsymbol{q}_h) : \nabla \boldsymbol{p} \; dx + \int_\Gamma [F^*(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+)] : [\boldsymbol{p}^- \otimes \boldsymbol{n}^- + \boldsymbol{p}^+ \otimes \boldsymbol{n}^+] \; ds = \int_\Omega \boldsymbol{S}(\boldsymbol{q}_h) \cdot \boldsymbol{p} \; dx, \tag{3.30}$$

where Γ is the set of interior facets. Since we have periodic boundary conditions, all facets are interior facets, and we do not need separate boundary terms.

One choice of numerical flux for the non-linear problem is the local Lax-Friedrichs flux [30], sometimes called the Rusanov flux, which is given by

$$F^*\left(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+\right) = F^{\mathrm{LF}}\left(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+\right) = \{\{F(\boldsymbol{q}_h)\}\} + \frac{1}{2}|\tau|\left(\boldsymbol{q}_h^- \otimes \boldsymbol{n}^- + \boldsymbol{q}_h^+ \otimes \boldsymbol{n}^+\right) \tag{3.31}$$

where $\boldsymbol{n}^\pm$ denotes the outward pointing unit normal of cell $K_\pm$,

$$\boldsymbol{q}_h^\pm = \lim_{\varepsilon \to 0} \boldsymbol{q}_h\left(\boldsymbol{x} - \varepsilon \boldsymbol{n}^\pm\right)$$

and τ is given by

$$\tau = \max\left\{\left|\boldsymbol{u}^- \cdot \boldsymbol{n}^-\right| + \sqrt{\phi_B + \phi_S^-},\ \left|\boldsymbol{u}^+ \cdot \boldsymbol{n}^+\right| + \sqrt{\phi_B + \phi_S^+}\right\}. \tag{3.32}$$

Note that here lowercase $\boldsymbol{u}$ is the velocity and is given by

$$\boldsymbol{u} = \frac{\boldsymbol{U}}{\phi_s + \phi_B}.$$

τ, given by equation (3.32), is the largest eigenvalue of the non-linear eigenproblem coming from the flux terms in equation (3.29): $F(\boldsymbol{q})\boldsymbol{n} = \tau \boldsymbol{q}$.

To investigate semi-implicit timestepping in a simplified context, in the next section we restrict consideration to the linearised SWE.

3.3.2 Linearised shallow water equations

In this section we consider the linearised SWE from section 2.2:

$$\frac{\partial \phi_S}{\partial t} + \nabla \cdot U = 0 \tag{3.33}$$

$$\frac{\partial U}{\partial t} + \nabla(\phi_B \phi_S) = -f(k \times U) + \phi_S \nabla \phi_B \tag{3.34}$$

By writing the flux of this system as

$$\begin{aligned} \overline{F} &= \begin{pmatrix} \boldsymbol{U}^\top \\ \phi_B \phi_S \mathcal{I}_2 \end{pmatrix} \\ &= \begin{pmatrix} U_1 & U_2 \\ \phi_B \phi_S & 0 \\ 0 & \phi_B \phi_S \end{pmatrix}, \end{aligned} \tag{3.35}$$

we can write equations (3.33) and (3.34) as a conservation law, like the general case in equation (3.23). Following the same steps as in section 3.3.1, we arrive at the following global system:

$$\int_\Omega \frac{\partial \boldsymbol{q}_h}{\partial t} \cdot \boldsymbol{p} - \overline{F}(\boldsymbol{q}_h) : \nabla \boldsymbol{p}\, dx + \int_\Gamma [\overline{F^*}(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+)] : [\boldsymbol{p}^- \otimes \boldsymbol{n}^- + \boldsymbol{p}^+ \otimes \boldsymbol{n}^+]\, ds = \int_\Omega \boldsymbol{S}(\boldsymbol{q}_h) \cdot \boldsymbol{p}\, dx \tag{3.36}$$

This equation only differs from equation (3.30) in the flux term $\overline{F}$ defined above and the numerical flux $\overline{F^*}$. For the numerical flux we consider two cases, an upwind flux as we saw in the advection example and the local Lax-Friedrichs flux we used in the non-linear

case.

That is $\overline{F^{\mathrm{LF}}} = F^{\mathrm{LF}}$, but the parameter τ in equation (3.31) simplifies to $\tau = \sqrt{\phi_B}$. This is because the non-linear eigenproblem we previously had to solve becomes the matrix eigenvalue problem:

$$\overline{F}(\boldsymbol{q})\boldsymbol{n} = n_x\overline{F}_x\boldsymbol{q} + n_y\overline{F}_y\boldsymbol{q} = \begin{pmatrix} 0 & n_x & n_y \\ n_x\phi_B & 0 & 0 \\ n_y\phi_B & 0 & 0 \end{pmatrix} \begin{pmatrix} \phi_S \\ U_1 \\ U_2 \end{pmatrix} = \tau\boldsymbol{q}$$

The $\overline{F}_x$ and $\overline{F}_y$ parts are seen again in the construction of the global system matrix in equation (3.38).

The upwind flux (defined in equation (3.13) for advection) is more complicated, but we can write the upwind numerical flux for the potential height and momentum fields separately as

$$\phi_S^{\mathrm{up}} = \{\{\boldsymbol{U}\}\} + \frac{1}{2}\sqrt{\phi_B}[\![\phi_S]\!]$$
$$\boldsymbol{U}^{\mathrm{up}} = \phi_B\{\{\phi_S\}\} + \frac{1}{2}\sqrt{\phi_B}[\![\boldsymbol{U}]\!]$$

Doing so, we can write the upwind flux

$$\overline{F^{\mathrm{up}}}(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+) = \begin{pmatrix} [\phi_S^{\mathrm{up}}]^\top \\ \boldsymbol{U}^{\mathrm{up}}\mathcal{I}_2 \end{pmatrix}. \tag{3.37}$$

Although the upwind and Lax-Friedrich fluxes look similar, they differ in the "jump" term. The Lax-Friedrich jump term has entries in all components of the 3×2 matrix, but the upwind flux has two zero entries, as the $\mathcal{I}_2$ identity is used.

This leads to some subtle differences. The most important advantage of the Lax-Friedrich flux is it's applicability to non-linear problems In fact the Lax-Friedrichs flux is one of the most widely used fluxes when using RKDG timestepping methods, since it is one of the simplest that can be used with non-linear problems. On of the disadvantages that we observe in section 4.1.1, is that the hybridised form of the Lax-Friedrich flux can only be reduced to two independent DOFs, whereas upwind can be reduced to one. This leads to a larger linear system to invert when using hybridisation.

Since there is no suitable equivalent of upwinding for the non-linear SWE, we exclusively use Lax-Friedrich in the non-linear case. Upwinding does have some advantages, one mentioned already, is the smaller hybridised linear system. If the problem is linear, this is a strong case for using upwind over Lax-Friedrichs. The DG method with Lax-Friedrichs numerical trace has also been observed [17] to be more dissipative than the DG method with the upwinding numerical trace for advection dominated problems.

The upwind and Lax-Friedrichs fluxes here have been chosen as they are both simple, suitable for the SWE and have hybridised forumations. However, these are not the only flues available and different problems call for different fluxes, for a comparison of the wide range available refer to the comparison by Qui et al. [43].

3.3.3 Matrix construction

To construct the matrix equation from the weak form of the shallow water equations is far more complicated than for the advection equation. In section 3.2.2 there was only one scalar field in the weak form, but for the shallow water equations we have two coupled fields, one scalar, one vector. These two fields are collected into the vector $\boldsymbol{q}_h$.

To write down the matrix equation, we need to know the degrees of freedom vector for the SWE. In the advection example, we collected all the degrees of freedom associated to the scalar valued function ϕ_h in a cell K together into a column vector

$$\underline{\phi}_K = (\phi_1, \dots, \phi_j, \dots)^\top,$$

and stacked all of these column vectors on top of one another to get

$$\underline{\phi} = (\underline{\phi}_{K_1}^\top, \dots, \underline{\phi}_{K_\ell}^\top, \dots)^\top.$$

Now suppose we do this for the x and y components of momentum, U_1 and U_2 respectively, to get the DOF vectors $\underline{U}_1$ and $\underline{U}_2$. Finally we stack the $\underline{\phi}$ and the two components $\underline{U}_1, \underline{U}_2$ together to get one big DOF vector $\underline{q}$:

$$\underline{q} = (\underline{\phi}^\top, \underline{U}_1^\top, \underline{U}_2^\top)^\top$$

This represents the vector of trial functions $\boldsymbol{q}_h = (\phi_S, U_1, U_2)^\top$ that appears in the weak form. Similarly, recall that the test function is $\boldsymbol{p} = (\psi, V_1, V_2)^\top$.

We proceed term by term through the weak form for the linear problem

$$\int_\Omega \frac{\partial \boldsymbol{q}_h}{\partial t} \cdot \boldsymbol{p} - \overline{F}(\boldsymbol{q}_h) : \nabla \boldsymbol{p} \; dx + \int_\Gamma [\overline{F^*}(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+)] : [\boldsymbol{p}^- \otimes \boldsymbol{n}^- + \boldsymbol{p}^+ \otimes \boldsymbol{n}^+] \; ds = \int_\Omega \boldsymbol{S}(\boldsymbol{q}_h) \cdot \boldsymbol{p} \; dx,$$

noting that we could do the same for the non-linear problem by making suitable approximations to the non-linear operators.

The term containing the time derivative is the simplest

$$\int_\Omega \frac{\partial \boldsymbol{q}_h}{\partial t} \cdot \boldsymbol{p} \; dx = \int_\Omega \begin{pmatrix} \partial_t \phi_h \\ \partial_t U_1 \\ \partial_t U_2 \end{pmatrix} \cdot \begin{pmatrix} \psi_S \\ V_1 \\ V_2 \end{pmatrix} \; dx = \int_\Omega \partial_t \phi \psi \; dx + \int_\Omega \partial_t U_1 V_1 \; dx + \int_\Omega \partial_t U_2 V_2 \; dx.$$

As we saw previously, this integral can be expressed using a mass matrix. We now have three of these, which act on the three fields ϕ_S, U_1, U_2 independently, so we can write this as the block matrix

$$\underline{\underline{\mathrm{M}}}\,\underline{q} = \left(\begin{array}{c|c|c} M & & \\ \hline & M & \\ \hline & & M \end{array}\right) \begin{pmatrix} \underline{\phi} \\ \underline{U_1} \\ \underline{U_2} \end{pmatrix}$$

where each M is a block diagonal mass matrix for each field and empty blocks are zero. For a detailed construction of the matrix M, refer back to section 3.1.

In the term containing $\overline{F}(\boldsymbol{q}_h)$, we can split the tensor $\overline{F}$ into two columns, correspond-

ing to the two spatial directions x and y, and do the same for the term $\nabla \boldsymbol{p}$

$$\begin{aligned}
\int_\Omega \overline{F}(\boldsymbol{q}_h) : \nabla \boldsymbol{p}\ dx &= \int_\Omega \overline{F_x}\boldsymbol{q}_h \cdot \partial_x \boldsymbol{p} + \overline{F_y}\boldsymbol{q}_h \cdot \partial_y \boldsymbol{p}\ dx \\
&= \int_\Omega \begin{pmatrix} U_1 \\ \phi_B \phi_S \\ 0 \end{pmatrix} \cdot \begin{pmatrix} \partial_x \psi \\ \partial_x V_1 \\ \partial_x V_2 \end{pmatrix} + \begin{pmatrix} U_2 \\ 0 \\ \phi_B \phi_S \end{pmatrix} \cdot \begin{pmatrix} \partial_y \psi \\ \partial_y V_1 \\ \partial_y V_2 \end{pmatrix}\ dx \\
&= \int_\Omega \begin{pmatrix} 0 & 1 & 0 \\ \phi_B & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \phi_S \\ U_1 \\ U_2 \end{pmatrix} \cdot \begin{pmatrix} \partial_x \psi \\ \partial_x V_1 \\ \partial_x V_2 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ \phi_B & 0 & 0 \end{pmatrix} \begin{pmatrix} \phi_S \\ U_1 \\ U_2 \end{pmatrix} \cdot \begin{pmatrix} \partial_y \psi \\ \partial_y V_1 \\ \partial_y V_2 \end{pmatrix}\ dx \\
&= \int_\Omega U_1 \partial_x \psi\ dx + \int_\Omega \phi_B \phi_S \partial_x V_1\ dx + \int_\Omega U_2 \partial_y \psi\ dx + \int_\Omega \phi_B \phi_S \partial_y V_2\ dx
\end{aligned} \tag{3.38}$$

Note that when we expand each function in equation (3.38) in terms of the basis on a cell K what remains is the integral

$$D_{x,K,ij} = \int_K e_j \partial_x e_i\ dx, \tag{3.39}$$

which we use to define the matrix $D_{x,K}$. Like the mass matrix on a cell K, the differentiation matrix does not depend on any degrees of freedom on neighbouring cells, so we can define the global differentiation matrix using the direct sum

$$D_x = \bigoplus_{K \in \mathcal{T}_h} D_{x,K}. \tag{3.40}$$

So for a DG method the differentiation matrix is also block diagonal, just like the mass matrix M. We can define $D_{y,K}$ and D_y in the same way as for the x derivative, by using equations (3.39) and (3.40). We can write the differentiation matrix for the global system in equation (3.38) as

$$\underline{\underline{D}}\,\underline{q} = \left(\begin{array}{c|c|c} 0 & D_x & D_y \\ \hline D_{B,x}^\top & 0 & 0 \\ \hline D_{B,y}^\top & 0 & 0 \end{array}\right) \begin{pmatrix} \underline{\phi} \\ \underline{U_1} \\ \underline{U_2} \end{pmatrix}, \tag{3.41}$$

where it is understood the the matrix D_B contains the data about the bathymetry from the function ϕ_B. More precisely, we construct a finite dimensional approximation $\phi_{B,h}$ to the function ϕ_B via

$$\phi_B(\boldsymbol{x}) \approx \phi_{B,h}(\boldsymbol{x}) = \sum_i B_i e_i(\boldsymbol{x}).$$

In order to scale the differentiation matrix in equation (3.41) correctly, we define a new matrix

$$D_{B,x,K,ij} = \int_K e_j(\phi_{B,h})(\partial_x e_i)\ dx, \qquad D_{B,x} := \bigoplus_{K \in \mathcal{T}_h} D_{B,x,K},$$

so that $D_{B,x}\underline{\phi}$ is the matrix equivalent of the continuum operation $\phi_B \partial_x \phi_S$.

In the integral containing the numerical flux we can again split the tensors into x and y components. To do this we can write $\overline{F^*} : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^{3\times 2}$ as the x and y components,

which are the columns of $\overline{F^*}$, namely $\overline{F^*_x}, \overline{F^*_y} : \mathbb{R}^3 \to \mathbb{R}^{3\times 2}$. It is also possible to write all the individual components of the 3×2 matrix:

$$\overline{F^*}(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+) = \left(\overline{F^*_x}(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+), \overline{F^*_y}(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+)\right) = \begin{pmatrix} \overline{F^*_x}(\phi_S) & \overline{F^*_y}(\phi_S) \\ \overline{F^*_x}(U_1) & \overline{F^*_y}(U_1) \\ \overline{F^*_x}(U_2) & \overline{F^*_y}(U_2) \end{pmatrix}$$

Using this we rewrite numerical flux integral as

$$\begin{aligned} \int_\Gamma [\overline{F^*}(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+)] : [\boldsymbol{p}^- \otimes \boldsymbol{n}^- + \boldsymbol{p}^+ \otimes \boldsymbol{n}^+] \; ds \\ &= \int_\Gamma \overline{F^*_x}(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+) \cdot (\boldsymbol{p}^- \otimes n_x^- + \boldsymbol{p}^+ \otimes n_x^+) \\ &\quad + \overline{F^*_y}(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+) \cdot (\boldsymbol{p}^- \otimes n_y^- + \boldsymbol{p}^+ \otimes n_y^+) \; ds \\ &= \int_\Gamma \begin{pmatrix} \overline{F^*_x}(\phi_S) \\ \overline{F^*_x}(U_1) \\ \overline{F^*_x}(U_2) \end{pmatrix} \cdot \left(\begin{pmatrix} \psi^- \\ V_1{}^- \\ V_2{}^- \end{pmatrix} n_x^- + \begin{pmatrix} \psi^+ \\ V_1{}^+ \\ V_2{}^+ \end{pmatrix} n_x^+ \right) \\ &\quad + \begin{pmatrix} \overline{F^*_y}(\phi_S) \\ \overline{F^*_y}(U_1) \\ \overline{F^*_y}(U_2) \end{pmatrix} \cdot \left(\begin{pmatrix} \psi^- \\ V_1{}^- \\ V_2{}^- \end{pmatrix} n_y^- + \begin{pmatrix} \psi^+ \\ V_1{}^+ \\ V_2{}^+ \end{pmatrix} n_y^+ \right) \; ds. \end{aligned} \tag{3.42}$$

It quickly becomes apparent that the matrix representation of equation (3.42) is not simple. We refer back to section 3.2.2 and specifically figure 3-5, which shows that the numerical flux term appears in the matrix where all the black spots are. We will have a similar structure here, but the flux term not only takes values from neighbouring cells, but also each of the three different fields ϕ_S, U_1, U_2. We will just write the matrix for $\overline{F*}$ as

$$\underline{\underline{\mathrm{F}}}\,\underline{q} = \left(\begin{array}{c|c|c} Q_{\phi\phi} & Q_{\phi U_1} & Q_{\phi U_2} \\ \hline Q_{U_1\phi} & Q_{U_1U_1} & Q_{U_1U_2} \\ \hline Q_{U_2\phi} & Q_{U_2U_1} & Q_{U_2U_2} \end{array}\right) \begin{pmatrix} \underline{\phi} \\ \underline{U_1} \\ \underline{U_2} \end{pmatrix}, \tag{3.43}$$

where each Q matrix here has the same structure as the matrix made up of all of the black spots in figure 3-5, correctly scaled to give the correct value for the flux in each field. Recall that the black spots in figure 3-5 gave rise to the off diagonal entries of the matrix and correspond to the coupled degrees of freedom.

It remains to find the structure of the matrix that arises from the integral containing the source term $\boldsymbol{S}$, the right-hand side of equation (3.30).

$$\begin{aligned} \int_\Omega \boldsymbol{S}(\boldsymbol{q}_h) \cdot \boldsymbol{p} \; dx &= \int_\Omega \begin{pmatrix} 0 \\ fU_2 + (\partial_x \phi_B)\phi_S \\ -fU_1 + (\partial_y \phi_B)\phi_S \end{pmatrix} \cdot \begin{pmatrix} \psi \\ V_1 \\ V_2 \end{pmatrix} \; dx \\ &= \int_\Omega \begin{pmatrix} 0 & 0 & 0 \\ \partial_x \phi_B & 0 & f \\ \partial_y \phi_B & -f & 0 \end{pmatrix} \begin{pmatrix} \phi_S \\ U_1 \\ U_2 \end{pmatrix} \cdot \begin{pmatrix} \psi \\ V_1 \\ V_2 \end{pmatrix} \; dx \end{aligned}$$

This term again only uses local degrees of freedom, so using what we have seen above, we

deduce that

$$\underline{\underline{\mathrm{S}}}\,\underline{q} = \left(\begin{array}{c|c|c} 0 & 0 & 0 \\ \hline M_{B,x} & 0 & fI \\ \hline M_{B,y} & -fI & 0 \end{array}\right) \begin{pmatrix} \underline{\phi} \\ \underline{U_1} \\ \underline{U_2} \end{pmatrix}, \tag{3.44}$$

where I is the appropriately sized identity matrix To scale the mass matrix in equation (3.41) correctly, we define a new matrix

$$M_{B,x,K,ij} = \int_K (\partial_x \phi_{B,h}) e_j e_i \, dx, \qquad M_{B,x,ij} := \bigoplus_{K \in \mathcal{T}_h} M_{B,x,K},$$

so that $M_{B,x}\underline{\phi}$ is the matrix equivalent of the continuum operation $\partial_x \phi_B \phi_S$.

The final matrix equation for the whole linear system is given by

$$\underbrace{\underline{\underline{\mathrm{M}}}\,(\partial_t \underline{q})}_{\text{Mass}} - \underbrace{\underline{\underline{\mathrm{D}}}\,\underline{q}}_{\text{Differentiation}} + \underbrace{\underline{\underline{\mathrm{F}}}\,\underline{q}}_{\text{Flux}} = \underbrace{\underline{\underline{\mathrm{S}}}\,\underline{q}}_{\text{Source}}. \tag{3.45}$$

3.3.4 Timestepping

At the start of this chapter we introduced the idea of semi-implicit timestepping schemes. We can write our non-linear system of equations as

$$\mathcal{M}(\partial_t \boldsymbol{q}_h, \boldsymbol{p}) + \mathcal{N}(\boldsymbol{q}_h, \boldsymbol{p}) = 0, \tag{3.46}$$

which is the same form as equation (3.25), but we have used $\mathcal{N}$ in place of $\mathcal{A}$ to make it clear that we have a non-linear operator.

It was shown in section 2.2 that the the linear part of the SWE contains the fast waves. Semi-implicit timestepping separates the fast waves ($\mathcal{L}$) and allows larger time steps to be taken, whilst maintaining stability. We do this by splitting the equation into the linear and non linear parts as follows

$$\mathcal{M}(\partial_t \boldsymbol{q}_h, \boldsymbol{p}) = -\underbrace{[\mathcal{N}(\boldsymbol{q}_h, \boldsymbol{p}) - \mathcal{L}(\boldsymbol{q}_h, \boldsymbol{p})]}_{\text{non-linear}}^{\text{explicit}} - \underbrace{[\mathcal{L}(\boldsymbol{q}_h, \boldsymbol{p})]}_{\text{linear}}^{\text{implicit}},$$

where $\mathcal{N}$ is the non-linear operator which we discuss in section 3.3.1, and $\mathcal{L}$ is the linear operator we discuss in section 3.3.2.

We can see the structure of the matrices which will arise from the application of timestepping from equation (3.45). Start by making the approximation $\partial_t \underline{q} = (\Delta t)^{-1}(\underline{q}^{(n+1)} - \underline{q}^{(n)})$. We write the matrix $\underline{\underline{\mathrm{L}}} = -\underline{\underline{\mathrm{D}}} + \underline{\underline{\mathrm{F}}} - \underline{\underline{\mathrm{S}}}$ and the denote non-linear vector function $\underline{\underline{\mathrm{N}}}\,(\cdot)$.

$$\underline{\underline{\mathrm{M}}} \left(\frac{\underline{q}^{(n+1)} - \underline{q}^{(n)}}{\Delta t} \right) = -\left[\underline{\underline{\mathrm{N}}}\,(\underline{q}^{(n)}) - \underline{\underline{\mathrm{L}}}\,\underline{q}^{(n)}\right] - \left[\underline{\underline{\mathrm{L}}}\,\underline{q}^{(n+1)}\right]$$

We only treat the linear terms implicitly. Gathering implicit terms $\underline{q}^{(n+1)}$ on the left and explicit terms $\underline{q}^{(n)}$ on the right, we obtain:

$$(\underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\mathrm{L}}}\,)\underline{q}^{(n+1)} = (\underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\mathrm{L}}}\,)\underline{q}^{(n)} - \Delta t \underline{\underline{\mathrm{N}}}\,(\underline{q}^{(n)}) \tag{3.47}$$

Here we can clearly see the matrix which is involved in the solve step, namely $(\underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\mathrm{L}}}\,)$.

The matrix $(\underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\mathrm{L}}}\,)$ contains a large number of off diagonal entries, which is precisely what makes solving difficult. In chapter 5 we demonstrate different methods for

solving problems involving this matrix. A Schur complement factorisation is used to take advantage of the large scale block structure of the matrix, and appropriate solvers and preconditioners are used on the sub-blocks. The exact procedure for this solve is outlined in chapter 7.

An alternative approach is to try and manipulate the original shallow water equations to get a different matrix. By changing the matrix we use in the solve, it is possible to compare the two alternate approaches to efficiently solving the SWE using a DG discretisation. In chapter 8, this is precisely what we do.

The next chapter, chapter 4, we provide an alternative DG discretisation, by using hybridisation. The purpose of hybridisation is to remove the off diagonal entries in the matrix that arise from coupled DOFs. This is done by coupling these DOFs to additional degrees of freedom that live on the facets of the underlying mesh. The matrix with fewer off diagonal entries has better block structure, which gives another competitive technique for solving the SWE.

CHAPTER 4

HYBRIDISATION AND TIMESTEPPING

Hybridisation was first studied in the 1960's for conforming finite element methods solving problems in linear elasticity [28]. By breaking the continuity between cells in the mesh and instead enforcing continuity using Lagrange multipliers, the number of coupled degrees of freedom is reduced when compared to the original global system. This is the base for the original hybridisation method for *conforming* finite elements.

This idea can be extended to discontinuous Galerkin methods very naturally. With DG we start with discontinuities between cells by the choice of a discontinuous function space. Movement of information between cells and controlling the size of the discontinuity in the DG method is achieved by using a numerical flux. Instead of using Lagrange multipliers to enforce continuity between cells, we can use them to represent the numerical flux between cells.

This gives rise to the hybridised discontinuous Galerkin (HDG) method that we will study in this chapter. We will start from the shallow water equations that we discretised in chapter 3 using the DG method and outline the HDG method.

4.1 Hybridisation

We look at hybridised DG method, because this method decouples the fields and reduces to a problem on the trace space, which has fewer degrees of freedom. Introducing an additional variable on the facets of the underlying mesh requires an additional equation to close the system and enforce the numerical flux. This extra equation leads to additional degrees of freedom, but decouples the ϕ_S and $\boldsymbol{U}$ variables, giving a better block matrix structure (see section 4.1.2 for details).

For the standard DG method the numerical flux introduces artificial diffusion terms, which correspond to the coupling between cells. By using the HDG method these artificial diffusion terms are eliminated and cells are now coupled via a trace variable. If we solve for the trace variable the global solution can be recovered using only cell local operations, since hybridisation eliminates the coupling between.

Recall the linear shallow water equations are:

$$\begin{aligned}\frac{\partial \phi_S}{\partial t} + \nabla \cdot U &= 0 \\ \frac{\partial U}{\partial t} + \nabla(\phi_B \phi_S) &= -f(k \times U) + \phi_S \nabla \phi_B \end{aligned} \tag{4.1}$$

By writing the flux of this system as

$$\begin{aligned}\overline{F} &= \begin{pmatrix} \boldsymbol{U}^\top \\ \phi_B \phi_S \mathcal{I}_2 \end{pmatrix} \\ &= \begin{pmatrix} U_1 & U_2 \\ \phi_B \phi_S & 0 \\ 0 & \phi_B \phi_S \end{pmatrix}.\end{aligned}$$

The SWE can be written as a conservation law and integrated over a single cell K to give

$$\int_K \frac{\partial \boldsymbol{q}_h}{\partial t} \cdot \boldsymbol{p} + (\nabla \cdot F(\boldsymbol{q}_h)) \cdot \boldsymbol{p} \, dx = \int_K \boldsymbol{S}(\boldsymbol{q}_h) \cdot \boldsymbol{p} \, dx.$$

So far we have done nothing different to the DG method in chapter 3.

The next step in the DG method is to integrate by parts and introduce a numerical flux $\overline{F^*}$. For the hybridised DG method, we will introduce the Lagrange multiplier $\widehat{\boldsymbol{q}} = \left(\widehat{\phi}_S, \widehat{\boldsymbol{U}}^\top\right)^\top \in \mathrm{Tr}(\mathcal{Q}_h)$ instead. See equation (3.24) for the definition of $\mathcal{Q}_h$. $\mathrm{Tr}(\mathcal{Q}_h)$ is the trace function space, which contains functions that live on the facets of the mesh that was used to $\mathcal{Q}_h$.

$$\begin{aligned}\mathrm{Tr}(\mathcal{Q}_h) &:= \{\widehat{\boldsymbol{q}} = \boldsymbol{q}_h|_\Gamma : \boldsymbol{q}_h \in \mathcal{Q}_h\} \\ &= \bigcup_{E \in \mathcal{E}(\Gamma)} \{\widehat{\boldsymbol{q}} \in [P^k(E)]^m\}\end{aligned}$$

where the notation is the same as in chapter 3.

To define the upwind numerical flux for the HDG method we define the variables

$$\begin{aligned}\widehat{\phi}_S^{\mathrm{up}} &= \boldsymbol{U} + \sqrt{\phi_B}(\phi_S - \widehat{\phi}_S)\boldsymbol{n} \\ \widehat{\boldsymbol{U}}^{\mathrm{up}} &= \phi_B \phi_S + \sqrt{\phi_B}(\boldsymbol{U} - \widehat{\boldsymbol{U}}) \cdot \boldsymbol{n},\end{aligned}$$

where $\widehat{\phi}_S \in \mathrm{Tr}(\Phi_h)$ and $\widehat{\boldsymbol{U}} \in \mathrm{Tr}(\mathcal{U}_h)$ and the solution values ϕ_S and $\boldsymbol{U}$ come from the cell K. With these variables we define the hybridised upwind flux as

$$\widehat{F}^{\mathrm{up}}(\boldsymbol{q}_h, \widehat{\boldsymbol{q}}) := \begin{pmatrix} [\widehat{\phi}_S^{\mathrm{up}}]^\top \\ \widehat{\boldsymbol{U}}^{\mathrm{up}} \mathcal{I}_2 \end{pmatrix}.$$

This is similar to the definition of DG upwind flux in equation (3.37), but rather than coupling the solution on each side of a facet ($\boldsymbol{q}^+$ and $\boldsymbol{q}^-$) with each other, we have coupled the local solution in a cell K ($\boldsymbol{q}$) to the trace variable $\widehat{\boldsymbol{q}}$.

An alternative is the hybridised Lax-Friedrichs numerical flux, which can be written as

$$\widehat{F}^{\mathrm{LF}}(\boldsymbol{q}_h, \widehat{\boldsymbol{q}}) := F(\boldsymbol{q}_h) + |\tau|(\boldsymbol{q}_h - \widehat{\boldsymbol{q}}) \otimes \boldsymbol{n}$$

where $\tau = \sqrt{\phi_B}$ for the linear SWE in equation (4.1). This is similar to the Lax-Friedrichs

flux we defined in equation (3.31).

For the HDG method, both fluxes $\widehat{F}^{\text{up}}$ and $\widehat{F}^{\text{LF}}$ are written in terms of the solution in the cell and the additional trace variable $\widehat{\boldsymbol{q}}$. This is instead of being defined in terms of the solution value on each side of the facet and coupling DOFs together directly as we did in the DG method. Figure 4-1 is a diagram of what is happening locally for a single mesh facet in both the DG and the hybridised DG method.

If we were to eliminate the new variable $\widehat{\boldsymbol{q}}$ by using the values from neighbouring cells, we would regain the upwind or Lax-Friedrich numerical flux $\overline{F^*}$ from the DG method. For the rest of the discussion wherever $\widehat{F}$ is used it should be understood that it can be replaced with one of $\widehat{F}^{\text{up}}$, $\widehat{F}^{\text{LF}}$ or any other suitable hybridised flux.

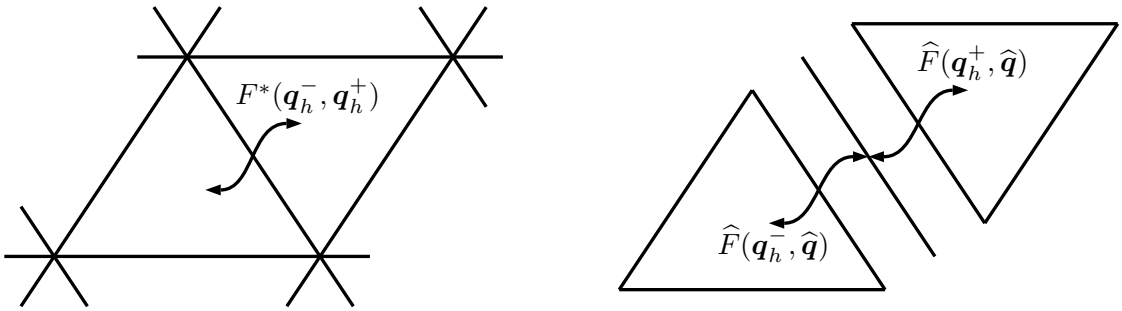

Figure 4-1: Numerical flux coupling 2 cells in the classical DG case compared to the numerical flux coupling both cells to a facet in the HDG case

Using a hybridised flux, and summing over all cells in our mesh, we have the new global system:

$$\int_\Omega \frac{\partial \boldsymbol{q}_h}{\partial t} \cdot \boldsymbol{p} - \overline{F}(\boldsymbol{q}_h) : \nabla \boldsymbol{p} \; dx + \int_\Gamma [\widehat{F}(\boldsymbol{q}_h, \widehat{\boldsymbol{q}})] : [\boldsymbol{p} \otimes \boldsymbol{n}] \; ds = \int_\Omega \boldsymbol{S}(\boldsymbol{q}_h) \cdot \boldsymbol{p} \; dx, \tag{4.2}$$

This is an equation of two unknowns $\boldsymbol{q}$ and $\widehat{\boldsymbol{q}}$, so we must introduce an equation to close the system. Since the the numerical flux has a unique value on each facet, we want

$$\left[\!\left[\widehat{F}(\boldsymbol{q}_h, \widehat{\boldsymbol{q}}) \right]\!\right] = 0, \tag{4.3}$$

which we enforce weakly using

$$\int_\Gamma \left[\!\left[\widehat{F}(\boldsymbol{q}_h, \widehat{\boldsymbol{q}}) \right]\!\right] \cdot \widehat{\boldsymbol{p}} \; ds = 0$$

to close the system.

If we can decompose the hybridised numerical flux into two parts

$$\widehat{F}(\boldsymbol{q}_h, \widehat{\boldsymbol{q}}) = \widehat{F}(\boldsymbol{q}_h) + \widehat{F}(\widehat{\boldsymbol{q}})$$

the first only involving the original unknown $\boldsymbol{q}_h$, the second only involving the trace unknown $\widehat{\boldsymbol{q}}$, then the hybridised problem can also be written as the bilinear forms in equation (4.4). The hybridised weak form of the shallow water equations is find $\boldsymbol{q}_h \in \mathcal{Q}_h$ and $\widehat{\boldsymbol{q}} \in \text{Tr}(\mathcal{Q}_h)$ such that

$$\begin{aligned} \mathcal{M}(\partial_t \boldsymbol{q}_h, \boldsymbol{p}) + \widehat{\mathcal{A}}(\boldsymbol{q}_h, \boldsymbol{p}) + \mathcal{G}(\widehat{\boldsymbol{q}}, \boldsymbol{p}) &= 0 \\ \mathcal{G}^\top(\boldsymbol{q}_h, \widehat{\boldsymbol{p}}) + \mathcal{T}(\widehat{\boldsymbol{q}}, \widehat{\boldsymbol{p}}) &= 0, \end{aligned} \tag{4.4}$$

holds for all $\boldsymbol{p} \in \mathcal{Q}_h$ and $\widehat{\boldsymbol{p}} \in \mathrm{Tr}(\mathcal{Q}_h)$. Here $\widehat{\mathcal{A}}$ is a hybridised form of $\mathcal{A}$ in equation (3.25). That is

$$\widehat{\mathcal{A}}(\boldsymbol{q}_h, \boldsymbol{p}) := \int_\Gamma [\widehat{F}(\boldsymbol{q}_h)] : [\boldsymbol{p} \otimes \boldsymbol{n}] \; ds - \int_\Omega \overline{F}(\boldsymbol{q}_h) : \nabla \boldsymbol{p} + \boldsymbol{S}(\boldsymbol{q}_h) \cdot \boldsymbol{p} \; dx,$$

where $\overline{F}$ is defined in equation (3.35) for the DG discretisation of the SWE. We define

$$\mathcal{G}(\widehat{\boldsymbol{q}}, \boldsymbol{p}) := \int_\Gamma [\widehat{F}(\widehat{\boldsymbol{q}})] : [\boldsymbol{p} \otimes \boldsymbol{n}] \; ds$$

which represents the coupled numerical flux terms that involve the variable $\widehat{\boldsymbol{q}}$. The adjoint of $\mathcal{G}$ is

$$\mathcal{G}^\top(\boldsymbol{q}_h, \widehat{\boldsymbol{p}}) = \int_\Gamma [\![\widehat{F}(\boldsymbol{q}_h)]\!] \cdot \widehat{\boldsymbol{p}} \; ds$$

and

$$\mathcal{T}(\widehat{\boldsymbol{q}}, \widehat{\boldsymbol{p}}) := 2 \int_\Gamma [\widehat{F}(\widehat{\boldsymbol{q}})\boldsymbol{n}] \cdot \widehat{\boldsymbol{p}} \; ds.$$

It is possible to simplify the system further by eliminating some of the trace degrees of freedom, as we see in the next section.

4.1.1 Reducing trace variables

Equation (4.4) makes use of the trace variable $\widehat{\boldsymbol{q}}$, which is composed of three trace variables $\widehat{\phi}_S$ and the two components of $\widehat{\boldsymbol{U}}$. It is shown by Bui-Thanh [15] that the number of trace variables required can be reduced to 1 or 2 when using the upwind and Lax-Friedrich flux respectively.

We start with the upwind flux, which we can multiply with a facet normal to obtain

$$\widehat{F}^{\mathrm{up}}\boldsymbol{n} = \begin{pmatrix} (\boldsymbol{U} \cdot \boldsymbol{n}) + \sqrt{\phi_B}(\phi_S - \widehat{\phi}_S) \\ \phi_B \phi_S \boldsymbol{n} + \sqrt{\phi_B}(\boldsymbol{U} \cdot \boldsymbol{n} - \widehat{\boldsymbol{U}} \cdot \boldsymbol{n})\boldsymbol{n} \end{pmatrix}. \tag{4.5}$$

Recall that the hybridised flux is defined in terms of local degrees of freedom, this means ϕ_S and $\boldsymbol{U}$ in equation (4.5) are the solution on a given cell K. Only when we consider the problem on the trace space (which involves the $[\![\;]\!]$ operation) do we have to distinguish the solution on each side of a facet.

Lemma 4.1.1 (Lemma 2.1 and Corollary 2.2 in [15])**.** *The hybridised upwind flux in equation* (4.5) *is equivalent to*

$$\widehat{F}^{up}\boldsymbol{n} = \begin{pmatrix} (\boldsymbol{U} \cdot \boldsymbol{n}) + \sqrt{\phi_B}(\phi_S - \widehat{\phi}_S) \\ \phi_B \widehat{\phi}_S \boldsymbol{n} \end{pmatrix}$$

where the trace variable $\widehat{\boldsymbol{U}} \in \mathrm{Tr}(\mathcal{U}_h)$ in the second row can be eliminated, reducing the number of independent trace variables to one.

Proof. To see the this is true we show that

$$(\boldsymbol{U} - \widehat{\boldsymbol{U}}) \cdot \boldsymbol{n} = -\sqrt{\phi_B}(\phi_S - \widehat{\phi}_S). \tag{4.6}$$

By expanding equation (4.3) we obtain the system of two equations

$$\begin{aligned}\boldsymbol{U}^- \cdot \boldsymbol{n}^- + \boldsymbol{U}^+ \cdot \boldsymbol{n}^+ + \sqrt{\phi_B}(\phi_S^- + \phi_S^+) - 2\sqrt{\phi_B}\widehat{\phi}_S = 0 \\ \phi_B(\phi_S^- \boldsymbol{n}^- + \phi_S^+ \boldsymbol{n}^+) + \sqrt{\phi_B}((\boldsymbol{U}^- \cdot \boldsymbol{n}^-)\boldsymbol{n}^- + (\boldsymbol{U}^+ \cdot \boldsymbol{n}^+)\boldsymbol{n}^+) \\ -\sqrt{\phi_B}((\widehat{\boldsymbol{U}} \cdot \boldsymbol{n}^-)\boldsymbol{n}^- + (\widehat{\boldsymbol{U}} \cdot \boldsymbol{n}^+)\boldsymbol{n}^+) = 0.\end{aligned}$$

These can be rearranged and written in terms of the jump and average

$$\begin{aligned}\widehat{\phi}_S &= \{\{\phi_S\}\} + \frac{1}{2\sqrt{\phi_B}}[\![\boldsymbol{U}]\!] \\ \widehat{\boldsymbol{U}} &= \{\{\boldsymbol{U}\}\} + \frac{\sqrt{\phi_B}}{2}[\![\phi_S]\!].\end{aligned} \tag{4.7}$$

We can evaluate the expressions $\phi_S - \widehat{\phi}_S$ and $\boldsymbol{U} - \widehat{\boldsymbol{U}}$ from each side of the facet. Since we get the same result both sides, without loss of generality we take the values on the "$-$" side:

$$\begin{aligned}\phi_S^- - \widehat{\phi}_S &= \phi_S^- - \left(\frac{\phi_S^- + \phi_S^+}{2}\right) - \frac{1}{2\sqrt{\phi_B}}[\![\boldsymbol{U}]\!] \\ &= \frac{\phi_S^- - \phi_S^+}{2} - \frac{1}{2\sqrt{\phi_B}}[\![\boldsymbol{U}]\!] \\ &= \frac{1}{2}(\phi_S^- \boldsymbol{n}^- + \phi_S^+ \boldsymbol{n}^+) \cdot \boldsymbol{n}^- - \frac{1}{2\sqrt{\phi_B}}[\![\boldsymbol{U}]\!] \\ &= \frac{1}{2}[\![\phi_S]\!] \cdot \boldsymbol{n}^- - \frac{1}{2\sqrt{\phi_B}}[\![\boldsymbol{U}]\!]\end{aligned}$$

$$\begin{aligned}(\boldsymbol{U}^- - \widehat{\boldsymbol{U}}) \cdot \boldsymbol{n}^- &= \boldsymbol{U}^- \cdot \boldsymbol{n}^- - \left(\frac{\boldsymbol{U}^- \cdot \boldsymbol{n}^- - \boldsymbol{U}^+ \cdot \boldsymbol{n}^-}{2}\right) - \frac{\sqrt{\phi_B}}{2}[\![\phi_S]\!] \cdot \boldsymbol{n}^- \\ &= \frac{\boldsymbol{U}^- \cdot \boldsymbol{n}^- - \boldsymbol{U}^+ \cdot \boldsymbol{n}^-}{2} - \frac{\sqrt{\phi_B}}{2}[\![\phi_S]\!] \cdot \boldsymbol{n}^- \\ &= \frac{1}{2}[\![\boldsymbol{U}]\!] - \frac{\sqrt{\phi_B}}{2}[\![\phi_S]\!] \cdot \boldsymbol{n}^-\end{aligned}$$

Comparing these two expressions we see that equation (4.6) holds. We can substitute equation (4.6) into the second row of equation (4.5). Doing so eliminates the trace variable $\widehat{\boldsymbol{U}}$. QED

The exact same procedure can be performed on the Lax-Friedrich flux:

$$\widehat{F}^{\mathrm{LF}}\boldsymbol{n} = \begin{pmatrix}\sqrt{\phi_B}(\boldsymbol{U} \cdot \boldsymbol{n}) + (\phi_S - \widehat{\phi}_S) \\ \sqrt{\phi_B}\phi_S \boldsymbol{n} + (\boldsymbol{U} - \widehat{\boldsymbol{U}})\end{pmatrix} \tag{4.8}$$

Notice the difference between equations (4.5) and (4.8) is that in the second row the Lax-Friedrichs flux has no $\boldsymbol{U} \cdot \boldsymbol{n}$ terms, only $\boldsymbol{U}$.

Lemma 4.1.2 (Lemma 2.4 in [15])**.** *The hybridised upwind flux in equation* (4.8) *is equivalent to*

$$\widehat{F}^{LF}\boldsymbol{n} = \begin{pmatrix}\widehat{\boldsymbol{U}} \cdot \boldsymbol{n} \\ \phi_B\phi_S \boldsymbol{n} + \sqrt{\phi_B}(\boldsymbol{U} - \widehat{\boldsymbol{U}})\end{pmatrix}$$

where the trace variable $\widehat{\phi}_S \in \mathrm{Tr}(\Phi_h)$ in the first row can be eliminated, reducing to one trace variable $\widehat{\boldsymbol{U}}$ with two degrees of freedom $(\widehat{U}_1, \widehat{U}_2)$.

Proof. The proof is identical to the proof of lemma 4.1.1. We expand equation (4.3) and obtain the same expressions as in equation (4.7), which give the same relation as equation (4.6). The only difference is $(\boldsymbol{U} - \widehat{\boldsymbol{U}}) \cdot \boldsymbol{n}$ does not appear in equation (4.8), so we substitute equation (4.6) into the first line. This gives us the result. QED

Corollary 4.1.3. *If we substitute equation* (4.7) *into equation* (4.5) *we recover the original DG upwind flux in equation* (3.37)*. If we substitute equation* (4.7) *into equation* (4.8) *we recover the original DG Lax-Friedrichs flux in equation* (3.31)*. This means that we are solving equivalent systems whether we choose to hybridise the system or not. The solution obtained using the DG method will be equal to the solution obtained using the HDG method for the same problem.*

From this point on we will refer to the trace variables in generality as $\lambda, \mu \in \Lambda_h$ as the trial and test functions respectively. If we are using the upwind numerical flux $\Lambda_h = \mathrm{Tr}(\Phi_h)$ and if we are using the Lax-Friedrich flux $\Lambda_h = \mathrm{Tr}(\mathcal{U}_h)$.

Because of the difference in the number of trace variables, we have to solve different trace space problems depending on whether we use the upwind or Lax-Friedrichs flux.

The reduced trace problem, which is derived from equation (4.4) can be written as the abstract bilinear form

$$a_{\mathrm{SWE}}(\lambda, \mu) = a_0(\lambda, \mu) + a_1(\lambda, \mu). \tag{4.9}$$

The procedure for obtaining this bilinear form is given in section 6.2.

In equation (4.9), the bilinear form a_0 is elliptic and is similar form to the one derived in Cockburn et al. [19] for a mixed Poisson problem,

$$a_0(\lambda, \mu) \longleftrightarrow a_{\mathrm{POISSON}}(\lambda, \mu),$$

albeit for different *lifting operators* (section 6.2.3). The bilinear form a_1 is an additional term that comes from the SWE, but we know that it is a strictly positive bilinear form.

Developing a geometric multigrid scheme for equation (4.9) is the main mathematical result for this thesis. This is not a simple task, since equation (4.9) is posed on the facets of the underlying mesh, and λ and μ only have support on the mesh skeleton.

The same problem posed using test and trial functions in the function space P^1 is much simpler as there is established theory for constructing a geometric multigrid scheme, since the trace space problem is elliptic. Rather than try and construct an entire grid hierarchy on the skeleton mesh, the aim will be to create a non-nested multigrid scheme that "restricts" functions $\lambda \in \mathrm{Tr}(\Phi_h)$ to a P^1 space, when using the upwind flux.

This involves determining what the equivalent coarse space problem is on the P^1 space. Once this is established, standard multigrid theory applies to the coarse space problem and a geometric multigrid algorithm can be performed. Furthermore, there are established multigrid results that can be applied to guarantee convergence in optimal $O(n)$ steps. The solution to the coarse space problem can be "prolongated" back to $\mathrm{Tr}(\Phi_h)$, and the solution to the SWE recovered using the hybridisation reconstruction algorithm (see chapter 7).

For the Lax-Firedrich flux where $\lambda \in \mathrm{Tr}(\mathcal{U}_h)$, the coarse space must be vector valued. We use a lowest order Raviart-Thomas space RT^1 for this purpose, but otherwise the procedure is the same as for the upwind case.

We will not try to derive the trace space problem posed in equation (4.9) here, as doing so is involved and technical, but the techniques involved are abstractly outlined in section 6.2 as well as the paper by Cockburn et al. [19]. Multigrid solvers are introduced

in section 5.4, where the mutigrid algorithm is written in terms systems of linear equations posed on regular grids. The theory behind multigrid is discussed further in section 6.1, where the multigrid algorithm is reformulated in terms of bilinear forms and function spaces. Using the multigrid algorithm written in terms of bilinear forms, the non-nested multigrid used on equation (4.9) is outlined in section 6.3.

4.1.2 Matrices

Following the same procedure as in section 3.3.3, we can write equation (4.4) as a matrix system. Since it is only the numerical flux term that has changed the matrices $\underline{\underline{\mathrm{M}}}, \underline{\underline{\mathrm{D}}}$ and $\underline{\underline{\mathrm{S}}}$ do not change, only $\underline{\underline{\mathrm{F}}}$. Having split $\widehat{F}(\boldsymbol{q}_h, \lambda) = \widehat{F}(\boldsymbol{q}_h) + \widehat{F}(\lambda)$ we can write the facet integral in equation (4.2) as

$$\begin{aligned}\int_\Gamma [\widehat{F}(\boldsymbol{q}_h)] : [\boldsymbol{p} \otimes \boldsymbol{n}] \; ds &= \int_\Gamma [\widehat{F}(\boldsymbol{q}_h)\boldsymbol{n}] \cdot \boldsymbol{p} \; ds \\ &= \int_\Gamma \begin{pmatrix} \widehat{F}(\phi_S) \\ \widehat{F}(U_1) \\ \widehat{F}(U_2) \end{pmatrix} \cdot \begin{pmatrix} \psi \\ V_1 \\ V_2 \end{pmatrix} \boldsymbol{n} \; ds.\end{aligned}$$

Looking at equation (4.5) or equation (4.8), we see that both the ϕ_S and $\boldsymbol{U}$ components of the hybridised flux depend on both ϕ_S and $\boldsymbol{U}$, so the matrix for this term can be written

$$\underline{\underline{\widehat{\mathrm{F}}}}\,\underline{q} = \left(\begin{array}{c|c|c} \widehat{Q}_{\phi\phi} & \widehat{Q}_{\phi U_1} & \widehat{Q}_{\phi U_2} \\ \hline \widehat{Q}_{U_1\phi} & \widehat{Q}_{U_1U_1} & \widehat{Q}_{U_1U_2} \\ \hline \widehat{Q}_{U_2\phi} & \widehat{Q}_{U_2U_1} & \widehat{Q}_{U_2U_2} \end{array}\right) \left(\begin{array}{c} \underline{\phi} \\ \hline \underline{U_1} \\ \hline \underline{U_2} \end{array}\right).$$

This matrix looks similar to equation (3.43), however since $\widehat{F}(\boldsymbol{q}_h)$ only depends on local degrees of freedom each of the $\widehat{Q}$ in the above matrix will be block diagonal.

Similarly to section 3.3.4 we define the hybridised matrix $\underline{\underline{\widehat{\mathrm{L}}}} := -\underline{\underline{\mathrm{D}}} + \underline{\underline{\widehat{\mathrm{F}}}} - \underline{\underline{\mathrm{S}}}$, where $\underline{\underline{\mathrm{D}}}$ and $\underline{\underline{\mathrm{S}}}$ are defined in equations (3.41) and (3.44) respectively. Since the blocks of $\underline{\underline{\widehat{\mathrm{F}}}}$ are now block diagonal, the blocks of $\underline{\underline{\widehat{\mathrm{L}}}}$ and $\underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\widehat{\mathrm{L}}}}$ are also block diagonal. This can be seen in the global system matrix in the top 3×3 blocks of the HDG matrix in figure 4-2.

To finish writing the matrix system we must define matrices for the remaining bilinear forms in equation (4.4). We pick e_i a basis of $\mathcal{Q}_h$ and $\hat{e}_i$ a basis of Λ_h to define the matrices

$$\begin{aligned}\underline{\underline{\mathrm{G}}}_{ij} &:= \mathcal{G}(\hat{e}_j, e_i) \\ &= \int_\Gamma [\widehat{F}(\hat{e}_i)\boldsymbol{n}] \cdot e_j \; ds\end{aligned}$$

$$\begin{aligned}\underline{\underline{\mathrm{T}}}_{ij} &= \mathcal{T}(\hat{e}_j, \hat{e}_i) \\ &= 2\int_\Gamma [\widehat{F}(\hat{e}_j)\boldsymbol{n}] \cdot \hat{e}_i \; ds.\end{aligned}$$

Given that e_i is a basis for $(\Phi_h \times \mathcal{U}_h)$ it is far larger than the basis of the trace function space Λ_h, making $\underline{\underline{\mathrm{G}}}$ a tall thin matrix which corresponds to the $\phi_S\lambda, U_1\lambda$ and $U_2\lambda$ blocks of the hybridised DG matrix in figure 4-2. $\underline{\underline{\mathrm{G}}}^\top$ is a short and long matrix corresponding to the $\lambda\phi_S, \lambda U_1, \lambda U_2$ blocks in the same figure. That leaves $\underline{\underline{\mathrm{T}}}$ as the small square $\lambda\lambda$ block in the bottom right.

If we use the same timestepping scheme from section 3.3.4, the system of matrix equations we must now solve is

$$\begin{aligned}(\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}})\underline{q}^{(n+1)} + \Delta t\underline{\underline{\mathrm{G}}}\,\underline{\lambda} &= (\underline{\underline{\mathrm{M}}} + \Delta t\underline{\underline{\mathrm{L}}})\underline{q}^{(n)} - \Delta t\underline{\underline{\mathrm{N}}}\,(\underline{q}^{(n)}) \\ \Delta t\underline{\underline{\mathrm{G}}}^{\top}\underline{q}^{(n+1)} + \Delta t\underline{\underline{\mathrm{T}}}\,\underline{\lambda} &= \underline{0}.\end{aligned}$$

Writing as a single block matrix system we get

$$\left(\begin{array}{c|c} \underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}} & \Delta t\underline{\underline{\mathrm{G}}} \\ \hline \Delta t\underline{\underline{\mathrm{G}}}^{\top} & \Delta t\underline{\underline{\mathrm{T}}} \end{array}\right)\begin{pmatrix} \underline{q}^{(n+1)} \\ \underline{\lambda} \end{pmatrix} = \begin{pmatrix} (\underline{\underline{\mathrm{M}}} + \Delta t\underline{\underline{\mathrm{L}}})\underline{q}^{(n)} - \Delta t\underline{\underline{\mathrm{N}}}\,(\underline{q}^{(n)}) \\ \underline{0} \end{pmatrix}. \tag{4.10}$$

This increases the size of the global matrix, but it now has better block structure than the original DG global system matrix we saw in equation (3.47). From section 3.3.4 we know that we can solve the SWE if we solve a global matrix problem with the matrix $\underline{\underline{\mathrm{M}}} + \Delta t\underline{\underline{\mathrm{L}}}$. Recall that $\underline{\underline{\mathrm{M}}}$ has only entries on the block diagonal, but $\underline{\underline{\mathrm{L}}}$ contains off-diagonal entries. Compare this to equation (4.10) where $\widehat{\underline{\underline{\mathrm{L}}}}$ and hence $\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}}$ has block diagonal structure.

The better block structure in equation (4.10) has two important consequences: Firstly, when we construct a Schur complement factorisation, the Schur complement, which here is $S = \Delta t\underline{\underline{\mathrm{T}}} - (\Delta t)^2\underline{\underline{\mathrm{G}}}^{\top}(\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}})^{-1}\underline{\underline{\mathrm{G}}}$, can be explicitly constructed. This is because $\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}}$ is easy to invert[1]. Furthermore, the Schur complement is also a *sparse* matrix. Whilst it is still possible to construct the Schur complement in the standard DG case, it is expensive to do so as the $\underline{\underline{\mathrm{L}}}$ matrix has many coupled degrees of freedom and the resulting matrix is not sparse, so in practise it is not done.

The second advantage of the block structure is that it is inexpensive to reconstruct the global solution. This is again due to the absence of coupling between cells. This means that the solution can be reconstructed in each cell of the mesh separately. Or, in terms of matrix operations, each sub-block of the blocks of $\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}}$ can be inverted independently.

The nested block structure is readily apparent in the sparsity patterns shown in figure 4-2. The figure shows the sparsity patterns of the DG matrix, corresponding to $\underline{\underline{\mathrm{M}}} + \Delta t\underline{\underline{\mathrm{L}}}$ is compared to the HDG matrix:

$$\left(\begin{array}{c|c} \underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}} & \Delta t\underline{\underline{\mathrm{G}}} \\ \hline \Delta t\underline{\underline{\mathrm{G}}}^{\top} & \Delta t\underline{\underline{\mathrm{T}}} \end{array}\right)$$

[1] this is done in two stages, outlined in section 7.3

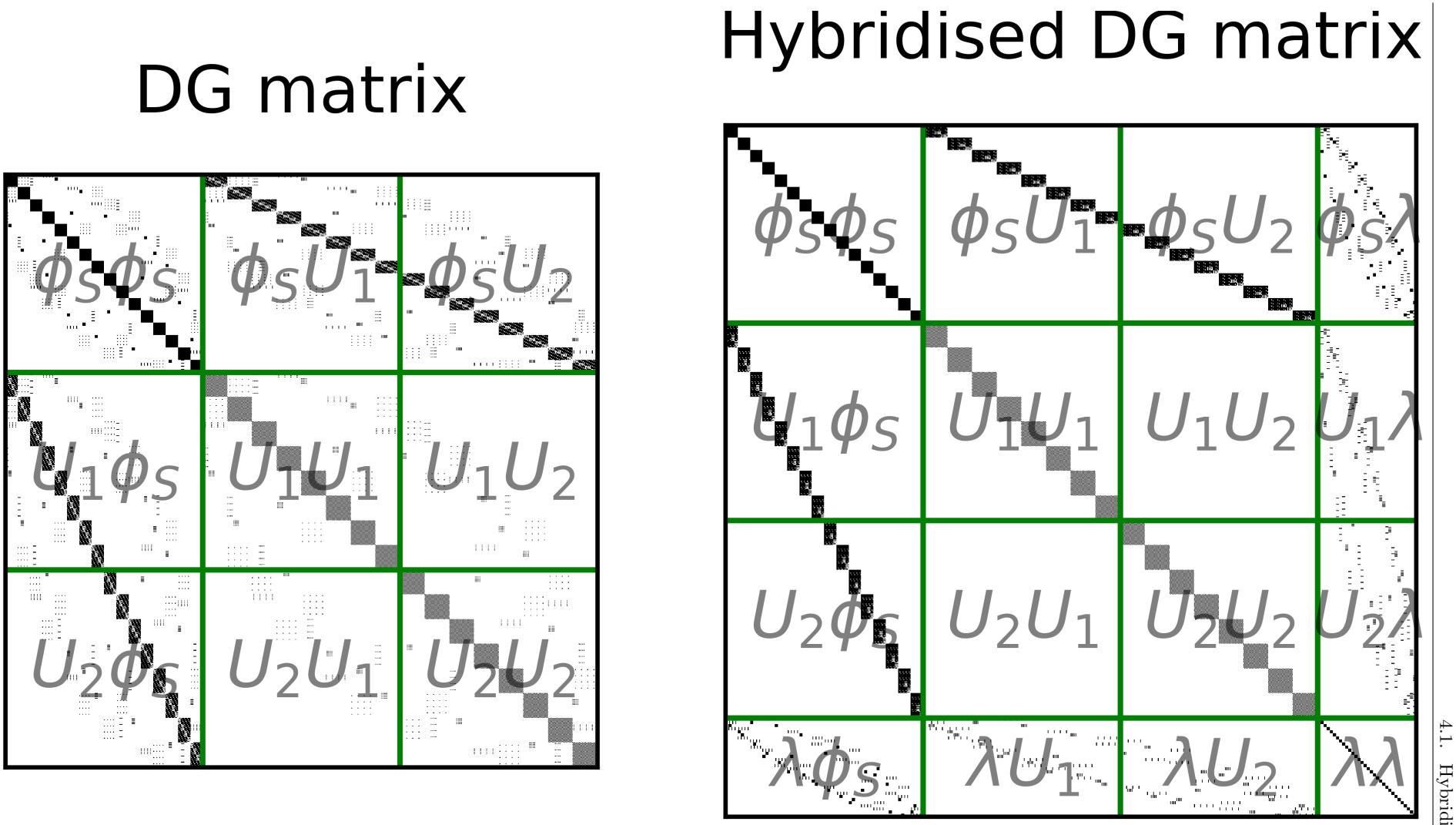

Figure 4-2: Sparsity pattern for DG and HDG matrices

Figure 4-2 shows the sparsity pattern for both methods when solving the SWE on a small (4×4) mesh of quadrilaterals, using a degree 3 DG method and upwind flux on the facets. The main difference to note is the off block diagonal entries are eliminated when using the HDG method. The cost of this elimination is the additional block in the matrix, but as the picture makes clear, the size of this block is very small when compared to the size of the full system matrix. A discussion of the relative block sizes and the Schur complement factorisation, which makes it possible to reconstruct the global solution by just performing a solve for the trace variable, is given in chapter 5.

Note in figure 4-2 the top-left 3×3 blocks *only* contain block matrices. These blocks correspond to cell local operations as all of the coupling is done using the trace variable. This means that when the solution $\phi_S, \boldsymbol{U}$ is reconstructed from λ we can consider the DOFs in each cell separately. This involves performing many *independent* dense matrix solves. These can be performed in parallel without communication between processes as each cell is independent. We see in section 7.3 that the Slate language we use in our implementation automatically forms these small dense matrices and performs the independent solves.

4.2 Timestepping

At the end of chapter 2 and at various points during the DG discetisation in chapter 3 the concept of timestepping has been used to demonstrate the form that global system matrices take when numerically solving PDEs and stepping them forward in time. So far simple explicit, implicit and IMEX have been demonstrated for the SWE and the three stage SSP RK3 method has been shown for the advection problem in section 3.2.3. In this section we review these timesteppers and introduce some more advanced and higher order timestepping schemes that we can use when solving time dependent problems.

After performing a DG spatial discretisation the problem that remains to be solved is a semi-discretised time dependent ordinary differential equation. In this section we will consider the general time dependent ODE

$$\frac{dq}{dt} = f(t, q), \qquad f(0, q) = q^{(0)} \tag{4.11}$$

where q is the quantity of interest and f is some function that depends on the time and the spatial variable q. To ensure that the method remains accurate, the size of the timestep may need to be limited.

If we assume f represents the discretisation of a PDE we can use the CFL condition to relate the timestep size to the grid spacing. This is a condition necessary for the convergence of the timestepping routine, named after Courant, Friedrichs and Lewy [22]. The CFL condition (we saw in equation (2.14)) is

$$C_{CFL} := u\frac{\Delta t}{\Delta x} < C.$$

Here Δx is the resolution of our spatial discretisation, Δt is the timestep size we use and u is the maximum wave speed in the problem we solve. If f is linear, the maximum waves speed can be found by finding the maximal eigenvalue of A, if $f(t, \underline{q}) = A\underline{q}$. The constant C varies depending on the timestepping method, but as a rough guide $C \approx 1$ for explicit timestepping methods and $C \approx 10$ for implicit timestepping methods.

One way of interpreting the CFL condition is a limit on how fast information can travel in a given timestepping scheme. In a single timestep the furthest any information can travel is $u\Delta t$. For the method to remain stable this distance is limited by some constant, C, times the spatial resolution. Information travelling faster than this is often realised as nonphysical behaviour in the solution, like wave amplitudes growing without bound and the solution "blowing up".

A brief review of standard explicit and implicit Runge-Kutta timestepping methods is given in appendix A.3. We proceed by looking at IMEX timestepping methods.

4.2.1 IMEX-RK

We look at methods that combine IMplicit and EXplicit (IMEX) methods. We start by looking at a slightly different general problem, instead of equation (4.11), we now consider

$$\frac{dq}{dt} = f(t, q) + g(t, q). \tag{4.12}$$

We have just split the right-hand side into two distinct functions. This allows us to treat f explicitly and g implicitly.

In the case of the semi-discretised SWE in equation (3.46) in section 3.3.4 we treat the

non-linear terms explicitly and the linear terms implicitly. This is done because the linear terms contain the fast waves in the SWE and these are stabilised by an implicit method. We see in this section that this splitting means we have to solve linear systems at each timestep and avoid having to perform a non-linear solve entirely.

The IMEX Runge-Kutta method combines together two Runge-Kutta methods, one based on an explicit timestepping scheme, the other based on an implicit one. An s stage IMEX-RK method for solving equation (4.12) is

$$q^{(n+1)} = q^{(n)} + \Delta t \sum_{i=1}^{s} b_i k_i + \Delta t \sum_{i=1}^{s} \bar{b}_i \bar{k}_i,$$

where

$$\bar{k}_i = g\left(t^{(n)} + \bar{c}_i \Delta t, \quad q^{(n)} + \Delta t \sum_{j=1}^{i-1} a_{ij} k_j + \Delta t \sum_{j=1}^{s} \bar{a}_{ij} \bar{k}_j\right) \quad \text{for } i = 1, \ldots, s \tag{4.13}$$

comes from an implicit method and

$$k_i = f\left(t^{(n)} + c_i \Delta t, \quad q^{(n)} + \Delta t \sum_{j=1}^{i-1} a_{ij} k_j + \Delta t \sum_{j=1}^{s} \bar{a}_{ij} \bar{k}_j\right) \quad \text{for } i = 1, \ldots, s$$

comes from an explicit method.

An IMEX-RK scheme can be compactly represented using two Butcher tableau, as shown in table 4.1 and examples are given in equations (4.14) to (4.16).

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & & \ddots & & \\ c_s & a_{s1} & a_{s2} & \cdots & a_{s\ s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \end{array} \qquad \begin{array}{c|cccc} \bar{c}_1 & \bar{a}_{11} & \bar{a}_{12} & \cdots & \bar{a}_{1s} \\ \bar{c}_2 & \bar{a}_{21} & \bar{a}_{22} & & \bar{a}_{2s} \\ \bar{c}_3 & \bar{a}_{31} & \bar{a}_{32} & & \bar{a}_{3s} \\ \vdots & \vdots & & \ddots & \vdots \\ \bar{c}_s & \bar{a}_{s1} & \bar{a}_{s2} & \cdots & \bar{a}_{ss} \\ \hline & \bar{b}_1 & \bar{b}_2 & \cdots & \bar{b}_s \end{array}$$

Table 4.1: Form of a dual Butcher tableau for IMEX RK timestepper

Although it is possible to perform timestepping with any implicit Runge-Kutta method as the implicit part of an IMEX method, we restrict ourselves to Diagonally Implicit Runge-Kutta (DIRK) methods outlined in appendix A.3.2. This IMEX algorithm is shown in algorithm 4.1.

Algorithm 4.1 only works for a DIRK implicit Butcher tableau and requires modification to work with other implicit timesteppers. Like algorithm A.1 we have a for loop that iterates over the number of stages. And like algorithm A.2 we have to perform a solve at each of these stages, but to save wasting computational effort we check whether the computed k_i actually appears on the right-hand side of equation (4.13) by checking whether $a_{ii} \neq 0$ before solving. Otherwise, we can calculate $\bar{k}_i$ the same way we do in an explicit step.

Furthermore, note that we must compute the implicit stage *before* the explicit stage so that we can include the $\underline{\bar{k}_i}$ in the explicit function evaluation.

We now look at three examples of IMEX methods. The first is the IMEX theta method

Algorithm 4.1: IMEX Runge-Kutta (DIRK only)

Input : $f : [0,T] \times \Omega \to \Omega$, $q^{(0)}$ – initial condition,
$(A, \underline{b}, \underline{c})$ – s-stage explicit Butcher tableau,
$(\bar{A}, \underline{\bar{b}}, \underline{\bar{c}})$ – s-stage DIRK Butcher tableau,
T – Total time, Δt – Timestep size

Output: $q^{(n)}$ for $n = 0, \ldots, N = \lceil \frac{T}{\Delta t} \rceil$

$t = 0$
$n = 0$
while $t < T$ **do**
  $\tilde{q} = q^{(n)}$
  for $i = 1, \ldots, s$ **do**
    $p = q^{(n)} + \Delta t \sum_{j=1}^{i-1} a_{ij} k_j + \Delta t \sum_{j=1}^{i-1} \bar{a}_{ij} \bar{k}_j$
    if $a_{ii} \neq 0$ **then**
      $\bar{k}_i = \text{Solve}\left(\tilde{k}_i = g(t^{(n)} + \bar{c}_i \Delta t, p + \Delta t \bar{a}_{ij} \tilde{k}_i)\right)$
    else
      $\bar{k}_i = g\left(t^{(n)} + \bar{c}_i \Delta t, p\right)$
    end
    $k_i = f\left(t^{(n)} + c_i \Delta t, q^{(n)} + \Delta t \sum_{j=1}^{i-1} a_{ij} k_j + \Delta t \sum_{j=1}^{i} \bar{a}_{ij} \bar{k}_j\right)$
    $\tilde{q} = \tilde{q} + \Delta t b_i k_i + \Delta t \bar{b}_i \bar{k}_i$
  end
  $q^{(n+1)} = \tilde{q}$
  $t = t + \Delta t$
  $n = n + 1$
end
return $q^{(n)}, n = 0, \ldots, N$

[34] in equation (4.14).

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1 & 0 \end{array} \qquad \begin{array}{c|cc} 0 & & \\ 1 & (1-\theta) & \theta \\ \hline & (1-\theta) & \theta \end{array}$$

IMEX Theta Method (4.14)

This method combines together explicit Euler for the explicit part and the theta method for the implicit part. The tableau for explicit Euler has been expanded to make it a two stage method, which is why it appears different to equation (A.4). It is possible to pad a method so that it has more stages than necessary, which means that a one stage method and a two stage method can be joined together to make an IMEX method, as we have done here. The parameter θ is not fixed and we are free to change it as we want.

The second method is the ARS2(2,3,2) method [8]. We introduce the same numbering introduced by Pareschi et al. [41] and also used in Weller et al. [55] for describing IMEX methods. The notation is [Name]$k(s, \sigma, p)$, where k is the order of the explicit method, s is the number of implicit stages (before padding), σ is the number of explicit stages (before padding) and p is the overall order of the IMEX scheme.

The ARS2(2,3,2) method is given by:

$$\gamma = 1 - \frac{1}{\sqrt{2}}, \qquad \delta = -\frac{2\sqrt{2}}{3}$$

$$\begin{array}{c|ccc} 0 & & & \\ \gamma & \gamma & & \\ 1 & \delta & (1-\delta) & \\ \hline & 0 & (1-\gamma) & \gamma \end{array} \qquad \begin{array}{c|ccc} 0 & 0 & & \\ \gamma & 0 & \gamma & \\ 1 & 0 & (1-\gamma) & \gamma \\ \hline & 0 & (1-\gamma) & \gamma \end{array}$$

IMEX ARS2(2,3,2) (4.15)

Where γ and δ are fixed parameters that ensure the accuracy of the timestepping method. This method combines together a second order, three stage explicit method, with a two stage DIRK method, which has been padded to create a three stage IMEX method. Notice there is only one non-zero on the diagonal of the implicit tableau, so we only require one solve per timestep using this method.

The third method is a strong stability preserving, SSP2(3,2,2), method [41], given by:

$$\begin{array}{c|ccc} 0 & & & \\ 1/2 & 0 & & \\ 1 & 0 & 1 & \\ \hline & 0 & 1/2 & 1/2 \end{array} \qquad \begin{array}{c|ccc} 1/2 & 1/2 & & \\ 0 & -1/2 & 1/2 & \\ 1 & 0 & 1/2 & 1/2 \\ \hline & 0 & 1/2 & 1/2 \end{array}$$

IMEX SSP2(3,2,2) (4.16)

This method combines together a second order, two stage explicit method (padded) with a three stage DIRK method to create a three stage IMEX method. This DIRK method has three non-zero coefficients on the diagonal, meaning there will be three solves every timestep.

A much more comprehensive analysis of timestepping methods for problems in numerical weather prediction is given by Weller et al. [55]. There are plenty of other IMEX timesteppers available, but these are the three we use to generate results in chapter 8.

4.2.2 IMEX for the SWE

We now look at the timestepping schemes we will use for the shallow water equations. Just as we did in section 3.3.4, we can define the global mass matrix $\underline{\underline{\mathrm{M}}}$, the rest of the system matrix $\underline{\underline{\mathrm{L}}} = -\underline{\underline{\mathrm{D}}} + \underline{\underline{\mathrm{F}}} - \underline{\underline{\mathrm{S}}}$ (the sum of the differentiation, flux and source matrices) and the denote non-linear part of the function $\underline{\underline{\mathrm{N}}}(\cdot)$. Before discretising in time, the matrix ODE that we propagate forward in time is

$$\frac{d}{dt}\left(\underline{\underline{\mathrm{M}}}\,\underline{q}\right) = \underline{\underline{\mathrm{N}}}(\underline{q}) + \underline{\underline{\mathrm{L}}}\,\underline{q}. \tag{4.17}$$

Unlike in the most general case, in equation (4.12), the right-hand side has no explicit time dependency.

For an IMEX-(DI)RK scheme, the equation we solve to obtain the solution at the next timestep is

$$\underline{\underline{\mathrm{M}}}\,\underline{q}^{(n+1)} = \underline{\underline{\mathrm{M}}}\,\underline{q}^{(n)} + \Delta t \sum_{i=1}^{s} b_i \underline{k}_i + \Delta t \sum_{i=1}^{s} \bar{b}_i \bar{\underline{k}}_i,$$

where we must perform an additional solve involving the mass matrix $\underline{\underline{\mathrm{M}}}$ to recover $\underline{q}^{(n+1)}$. This is not an issue as the DG mass solve is an inexpensive step. The $\bar{\underline{k}}_i$ are given by solving

$$(\underline{\underline{\mathrm{M}}} - \Delta t \bar{a}_{ii} \underline{\underline{\mathrm{L}}})\bar{\underline{k}}_i^* = \underline{\underline{\mathrm{M}}}\,\underline{q}^{(n)} + \Delta t \sum_{j=1}^{i-1} a_{ij}\underline{k}_j + \Delta t \sum_{j=1}^{i-1} \bar{a}_{ij}\bar{\underline{k}}_j \quad \text{for } i = 1, \ldots, s, \tag{4.18}$$

for $\bar{k}_i^*$ and storing $\bar{k}_i = \underline{\underline{\mathrm{L}}}\,\bar{k}_i^*$. Equation (4.18) is just a rearrangement of equation (4.13) to show the method involves a solve with the matrix $\underline{\underline{\mathrm{M}}} - \Delta t \bar{a}_{ii}\underline{\underline{\mathrm{L}}}$. This solve must be performed at each stage where $\bar{a}_{ii} \neq 0$. Otherwise a linear function evaluation is required to obtain $\bar{\underline{k}}_i$, that is, if $\bar{b}_i \neq 0$ or any of the $\bar{a}_{\ell i} \neq 0$ for any ℓ. The $\underline{k}_i$ are given by the explicit equation

$$\underline{k}_i = \underline{\underline{\mathrm{N}}}\left(\underline{q}^{(n)} + \Delta t \sum_{j=1}^{i-1} a_{ij}\underline{k}_j + \Delta t \sum_{j=1}^{i} \bar{a}_{ij}\bar{\underline{k}}_j\right) \quad \text{for } i = 1, \ldots, s,$$

which involved evaluating the non-linear function $\underline{\underline{\mathrm{N}}}$ for each stage where the $\underline{k}_i$ is used. A $\underline{k}_i$ is used if $b_i \neq 0$ or any of the $a_{\ell i} \neq 0$ for any ℓ.

	total order	evaluations		solves	
		$\underline{\underline{\mathrm{L}}}$	$\underline{\underline{\mathrm{N}}}$	$\underline{\underline{\mathrm{M}}}$	$\underline{\underline{\mathrm{L}}}$
timestepper					
Explicit Euler	1	1	1	1	0
Heun	2	2	2	2	0
SSPRK3	3	3	3	3	0
Theta θ =0.5	1	1	1	0	1
IMEXTheta θ =0.5	1	1	1	2	1
IMEXARS2(2,3,2)	2	2	3	2	2
IMEXSSP2(3,2,2)	2	3	2	1	3

Table 4.2: Total order, number of linear and non-linear function evaluations, and number of mass and system matrix solves required for all discussed timestepping methods

A summary of the number of linear and non-linear function evaluations and mass and system matrix solves required per timestep for all the different methods we implement is given in table 4.2. This allows us to determine which methods are likely to be more efficient, based on the fact that, in general a linear function evaluation is slightly less expensive than a non-linear function evaluation[2], function evaluation is less expensive than a DG mass matrix solve, and a DG system matrix solve is the most expensive operation.

We can reason that the timesteppers in table 4.2 are sorted roughly in terms of how expensive they are to timestep a DG method. This deduction is confirmed *for DG methods only* in table 8.9. However, the system matrix solve in equation (4.18) could be replaced with a HDG system matrix solve. With the solvers and preconditioners outlined in chapters 5 and 6 and the timesteppers in table 4.2, we will demonstrate that HDG combined with implicit timestepping can run in a similar time to a DG method with an explicit timestepper.

[2]in practise these times are comparable

CHAPTER 5

ELLIPTIC SOLVERS

In the previous chapters we focused on taking a PDE which describes a problem in a continuous space and transforming it into a system of linear equations using the DG and HDG methods of discretisation. In doing so we have made it possible to solve the problem on a computer using numerical methods. We have not yet given any procedure for actually solving the resulting matrix equation that arises, which is what this chapter is dedicated to.

All of the work so far has allowed us to find a solution to a PDE by solving a matrix equation

$$A\underline{x} = \underline{b}. \tag{5.1}$$

If we can calculate $\underline{x}$, which here represents a vector containing all of the degrees of freedom, then we can reconstruct the solution to the PDE as a function. We will often just refer to the mechanism for solving this equation as a "solver". Mathematically we can calculate A^{-1}, and our solution is just $\underline{x} = A^{-1}\underline{b}$.

Computationally this can be done *directly* using an LU factorisation of the matrix A. This is found using some variant of Gaussian elimination and obtaining the solution $\underline{x}$ using forward and back substitution. In practise, calculating the LU decomposition directly is very expensive. For a dense matrix $A \in \mathbb{R}^{n\times n}$ this is $O(n^3)$ cost, where n is the number of degrees of freedom in the discretisation. However, sparse matrix LU algorithms have a lower computational cost that is a function of the number of non-zero entries in the matrix. It is not usually feasible to invert matrices with an LU factorisation when they are as large as 10^6 DOFs that we will use, or 10^9 that are used by the Met Office. Furthermore, the LU decomposition is numerically unstable, meaning that rounding errors from the use of finite precision arithmetic on a computer will grow.

For a numerical weather prediction application it would never be realistic to invert the global matrix to solve a PDE. The sheer number of degrees of freedom used in a global atmosphere model means that calculating an LU factorisation alone would take orders of magnitude longer than an operational time scale[1]. Solutions are instead found using iterative solvers.

[1] For forecasting an operational time scale might be as short as 20 minutes

5.1 Iterative solvers

An alternative to solving equation (5.1) with a direct method (such as an LU factorisation) is to apply an iterative solver, which is a procedure that generates a sequence of approximations $\underline{x}_1, \underline{x}_2, \ldots$ to the exact solution $\underline{x}$. Some initial guess $\underline{x}_0$ is used as a starting point and after a certain number of iterations, or after some convergence criterion is met, the iterative procedure is halted and the last iterate $\underline{x}_k$ is close to the true solution $\underline{x}$.

We look at two types of iterative methods, relaxation methods and Krylov subspace methods.

Relaxation methods break down the matrix A into two parts

$$A = M + N$$

and construct an iterative method of the form

$$\underline{x}_{k+1} = M^{-1}(b - N\underline{x}_k).$$

Typically, M is chosen in such a way that it is computationally inexpensive to invert.

To characterise both the Jacobi method and Gauss-Seidel we spilt the matrix into *three* parts

$$A = L + D + U,$$

where L is strictly lower triangular, D is diagonal and U is strictly upper diagonal. At each step some linear combination of L, D, U is inverted, which is easier than inverting A and an iterative scheme is built around this. Examples of relaxation methods are the (weighted) Jacobi method and the Gauss-Seidel method (and its generalisation successive over relaxation). These methods converge slowly, but can be used as smoothers for multigrid. A brief overview of these is given below.

In contrast, Krylov subspace methods solve equation (5.1) by constructing an approximate solution in the Krylov subspace

$$\mathcal{K}_m(A, \underline{b}) := \{\underline{b}, A\underline{b}, A^2\underline{b}, \ldots, A^{m-1}\underline{b}\}.$$

An example of a Krylov subspace method is the conjugate gradient method, section 5.1.3, and we will also briefly discuss the more general MINRES and GMRES solvers.

5.1.1 Jacobi Method

The Jacobi method is based around the fact that a diagonal matrix is easy and computationally cheap to invert. After splitting the matrix A into three parts, we observe that if $\underline{x}$ is a solution to the original problem, then it satisfies

$$D\underline{x} = b - (L + U)\underline{x}.$$

This simple rearrangement of the original equation provides the base for an iterative method

$$\underline{x}_{k+1} = D^{-1}(b - (L + U)\underline{x}_k). \tag{5.2}$$

This recurrence relation is the Jacobi method. If $D^{-1}(L + U)$ is a contraction then the sequence $\underline{x}_k$ will converge to the fixed point $\underline{x}$. This is true when the spectral radius[2] of

[2]The spectral radius is the maximum absolute value of all the eigenvalues of a matrix A and is denoted $\rho(A)$

$D^{-1}(L+U)$ is less than 1.

There is also a weighted variant of the Jacobi method which takes a linear combination of the previous iterate and the next, weighted by the parameter ω:

$$\underline{x}_{k+1} = \omega D^{-1}(b - (L+U)\underline{x}_k) + (1-\omega)\underline{x}_k \tag{5.3}$$

The price that is paid by having such a simple and computationally inexpensive iteration step in equation (5.2), is that it may take an extremely large number of iterations for equation (5.2) to converge. Whilst the parameter ω gives some control over the rate of convergence for equation (5.3), in practice we will never use the Jacobi method as a solver since it converges too slowly. However, the method will be revisited in section 5.4 as a multigrid smoother.

5.1.2 Gauss-Seidel Method

The Gauss-Seidel method works around a slightly different rearrangement of the original equation. The solution $\underline{x}$ is also satisfies

$$(L+D)\underline{x} = b - U\underline{x}. \tag{5.4}$$

Whilst the inverse of $(L+D)$ is not as easily calculable as D, equation (5.4) can still be solved efficiently using back substitution as $L+D$ is a lower triangular matrix.

This leads to the recurrence relation

$$\underline{x}_{k+1} = (L+D)^{-1}(b - U\underline{x}_k),$$

which is the Gauss-Seidel method. Again the homogeneous part of the right-hand side must be a contraction, i.e we need $\rho((L+D)^{-1}U) < 1$, for the method to converge.

The Gauss-Seidel also has a generalisation, more usually referred to as successive over relaxation. Rather than weighting two iterations like in the Jacobi method, successive over relaxation is based around weighting the diagonal part of the matrix A and the fixed point of

$$(\omega L + D)\underline{x} = \omega b - [\omega U + (\omega - 1)D]\underline{x}.$$

This gives the recurrence relation

$$\underline{x}_{k+1} = (\omega L + D)^{-1}(\omega b - [\omega U + (\omega - 1)D]\underline{x}_k).$$

By picking $\omega = 1$ this method just becomes the Gauss-Seidel method. Changing the parameter ω changes the weighting of the diagonal entries in the original matrix A and so may change the rate of convergence of the method. However, picking the parameter ω to improve the rate of convergence can be difficult. Just like the Jacobi method convergence of SOR may be slow and neither Gauss-Seidel, nor SOR, will be used as solvers, but as smoothers in section 5.4.

Due to Gauss-Siedel requiring a back substitution step, it is an inherently serial process, and great care must be taken to perform a parallel implementation of this method. In contrast, Jacobi can be readily parallelised.

5.1.3 Conjugate Gradient Method

The previous iterative methods were all relaxation methods, which were based around breaking the matrix A up into L, D, U. The conjugate gradient method is a Krylov

subspace method. The *matrix* A is not required for this method, we just need to be able to apply its action. This makes it an attractive choice when using matrix free methods.

For an $n \times n$ matrix, Krylov subspace solvers (such as the conjugate gradient method) guarantee convergence to the exact solution in n steps or fewer. This is because the method builds a basis of the space $\mathbb{R}^n$ one vector at a time. However, this is only the case when using exact arithmetic, when solving on a computer numerical errors mean we no longer have this guarantee. In practice Krylov subspace solvers are powerful iterative methods and they provide suitably accurate solutions usually in far fewer than n iterations.

We outline in algorithm 5.1 precisely how the conjugate gradient method works. We have written out the CG method in its preconditioned form to avoid having to repeat the algorithm later. As in the weighted versions of the relaxation methods, preconditioning is used to improve the convergence of the method. By fixing $P = I$–the identity matrix, we recover the unpreconditioned CG method.

Since we are limited by finite precision arithmetic, we stop the iterations once a given tolerance ε is reached.

Algorithm 5.1: Preconditioned conjugate gradient method for solving $A\underline{x} = \underline{b}$

Input : $A, P \in \mathbb{R}^{n\times n},\ \underline{x}_0, \underline{b} \in \mathbb{R}^n,\ \varepsilon \in \mathbb{R}$
Output: $\underline{x} \in \mathbb{R}^n,\ r \in \mathbb{R},\ k \in \mathbb{N}$
Initial guess $\underline{x}_0$
$\underline{r}_0 = \underline{b} - A\underline{x}_0$
$\underline{q}_0 = P^{-1}\underline{r}_0$
$\underline{p}_0 = \underline{q}_0$
for $k \geq 0$ **do**
  $\beta_k = \frac{\langle \underline{r}_k,\ \underline{q}_k \rangle}{\langle \underline{p}_k,\ A\underline{p}_k \rangle}$
  $\underline{x}_{k+1} = \underline{x}_k + \beta_k \underline{p}_k$
  $\underline{r}_{k+1} = \underline{r}_k - \beta_k A\underline{p}_k$
  if $\|\underline{r}_{k+1}\| < \varepsilon$ **then** break;
  $\underline{q}_{k+1} = P^{-1}\underline{r}_{k+1}$
  $\alpha_{k+1} = \frac{\langle \underline{r}_{k+1},\ \underline{q}_{k+1} \rangle}{\langle \underline{r}_k,\ \underline{q}_k \rangle}$
  $\underline{p}_{k+1} = \underline{q}_{k+1} + \alpha_{k+1}\underline{p}_k$
end
return $\underline{x}_{k+1}, \|\underline{r}_{k+1}\|, k$

The CG method can only be used if the matrix A in equation (5.1) is symmetric positive definite (SPD). If it is, the number of iterations required to obtain a fixed accuracy is bounded by some constant times the square root of the condition number $\kappa(P^{-1/2}AP^{-1/2})$, where P must also be SPD. Here $\kappa(\cdot)$ denotes the condition number in the 2–norm, which for SPD matrices is equal to the ratio of the maximum and minimum eigenvalues in absolute value ($\max_n |\lambda_n|$). The theory [26] uses uses the condition number of the matrix $P^{-1/2}AP^{-1/2}$ but algorithm 5.1 has been written in such a way that only $P^{-1}A$ is required. The theory still holds since

$$\kappa(P^{-1/2}AP^{-1/2}) = \frac{\lambda_{\max}(P^{-1}A)}{\lambda_{\min}(P^{-1}A)}.$$

This theoretical bound on the number of iterations is often pessimistic and in practice the number of iterations may be fewer.

It can be shown [51] for the unpreconditioned CG algorithm that the error for each iteration is bounded by

$$\|\underline{x}_k - \underline{x}\|_A \leq 2\left(\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k + \left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^{-k}\right)^{-1} \|\underline{x}_0 - \underline{x}\|_A,$$

where $\|\cdot\|_A$ is the norm induced by the matrix A, k is the iteration number and $\kappa = \kappa(A)$. For the preconditioned algorithm, replace A with $P^{-1/2}AP^{-1/2}$. This allows us to estimate that for a matrix with $\kappa(A) = 100$ (which is typical for a discretisation of the SWE), we require around 60 iterations to reduce the error of the initial guess by a factor of 10^{-5}. Note that as κ increases, the number of iterations we require to get the same error reduction also increases. If we can estimate the condition number, we can estimate how many iterations it takes to reach *any* tolerance.

5.1.4 MINRES and GMRES

The CG method only works for a matrix A which is symmetric positive definite. However, there are methods for working on even more general classes of matrices. MINRES is another Krylov subspace method, which works on matrices that are symmetric and non-singular [40]. GMRES is an even more general method that can be applied to a matrix A that is non-singular [48].

The matrices arising from DG discretisations which we consider in chapter 3 are not guaranteed to be symmetric or positive definite, so we cannot always use the CG method. But we can use GMRES instead. Less is known theoretically about the rate of convergence for a general matrix, but if the matrix is not SPD then the number of iterations required for GMRES to converge is bounded by the condition number of the matrix times some constant that depends on the conditioning of the basis of eigenvectors [26]. Another way to bound the iteration count for GMRES is to look at the field of values, but we will not look at this case.

5.2 Schur complement factorisation

If the system matrix has block structure, a Schur complement factorisation can be used to reduce the amount of work the solver does. The factorisation can be derived by performing block-wise Gaussian elimination on the matrix A. If a matrix has the (2×2) block structure

$$A = \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} I & FH^{-1} \\ 0 & I \end{pmatrix} \begin{pmatrix} S & 0 \\ 0 & H \end{pmatrix} \begin{pmatrix} I & 0 \\ H^{-1}G & I \end{pmatrix}, \tag{5.5}$$

where S is the Schur complement and $S = E - FH^{-1}G$, then the inverse can be written as

$$A^{-1} = \begin{pmatrix} E & F \\ G & H \end{pmatrix}^{-1} = \begin{pmatrix} I & 0 \\ -H^{-1}G & I \end{pmatrix} \begin{pmatrix} S^{-1} & 0 \\ 0 & H^{-1} \end{pmatrix} \begin{pmatrix} I & -FH^{-1} \\ 0 & I \end{pmatrix}. \tag{5.6}$$

Solving equation (5.1) now only requires solving matrix–vector systems involving the matrices H and S, both of which are smaller than the original matrix A. For S and H to be invertable, they must be square matrices, but they do not have to be the same size. So we could have $A \in \mathbb{R}^{(m+n)\times(m+n)}$ with the blocks $E \in \mathbb{R}^{m\times m}$ and $H \in \mathbb{R}^{n\times n}$. F and G are rectangular matrices, but this causes no issues.

Unlike the iterative methods we have seen so far, which approximate the solution, the

Schur complement factorisation in equation (5.6) holds exactly. If we invert S and H exactly, we will get the exact inverse of A. For very large matrices, such as the finite element matrices that arise when numerically solving a PDE on a fine grid, these block matrices are still too large to consider trying to invert using a direct method.

Writing out the factorisation is useful mathematically, but does not illustrate the work saved as effectively as algorithm 5.2 does. If we want to solve $A\underline{x} = \underline{b}$, which we can write in block form as

$$\begin{pmatrix} E & F \\ G & H \end{pmatrix} \begin{pmatrix} \underline{y} \\ \underline{z} \end{pmatrix} = \begin{pmatrix} \underline{f} \\ \underline{g} \end{pmatrix}, \tag{5.7}$$

we can use algorithm 5.2 to obtain the solution $\underline{x}$.

Algorithm 5.2: Schur complement factorisation method for solving $A\underline{z} = \underline{b}$

Input : $E \in \mathbb{R}^{m\times m},\ F \in \mathbb{R}^{m\times n},\ G \in \mathbb{R}^{n\times m}, H \in \mathbb{R}^{n\times n}$,
$\underline{f} \in \mathbb{R}^m,\ \underline{g} \in \mathbb{R}^n$
Output: $\underline{y} \in \mathbb{R}^m,\ \underline{z} \in \mathbb{R}^n$
Solve($H\underline{z}_1 = g$)
$S = E - FH^{-1}G$
Solve($S\underline{y} = \underline{f} - F\underline{z}_1$)
Solve($H\underline{z}_2 = -G\underline{y}$)
$\underline{z} = \underline{z}_1 + \underline{z}_2$
return $\underline{y},\ \underline{z}$

As we have already stated, if these blocks are large, it will be very expensive to invert them exactly using an LU factorisation. As an alternative we can use a suitable Krylov subspace solver instead of inverting the blocks. This idea is very powerful and introduces the concept of composable solvers. Once we have performed a Schur complement factorisation, we must solve another smaller matrix–vector system, which we can do using another preconditioned Krylov subspace method. We are then free to perform another, perhaps different, factorisation of the smaller matrix–vector system. We return to the idea of nesting different solvers together in chapter 7, section 7.4, when discussing the implementation of our solvers using PETSc.

Another approach we can take is to make an approximation to a given block. For instance, in algorithm 5.2 there are three steps that involve solving with the matrix H. If we could approximate H by another matrix that is easier to invert, then only expensive step is the solve involving the matrix S, where we can still use a Krylov subspace solver. For instance, we could use a diagonal or block diagonal approximation to H. It's worth noting that if we don't approximate H with a diagonal (or block diagonal) matrix, its inverse, and as a consequence the matrix S, will be dense, even if H is otherwise sparse.

We can now think about this in terms of the DG discretisation of a PDE. Recalling figures 3-4 and 3-5, the entries off the block diagonal come from the numerical flux and that these were essential for making the DG method work. Approximating a block of a matrix arising from the DG method by just removing off-diagonal entries will not be good enough to solve the original problem. However, this matrix with a diagonal approximation may be a suitable preconditioner as making this approximation we have made calculating the approximate inverse very cheap.

5.2.1 Preconditioning

Whilst using an iterative method will eventually yield a solution, we seek a method which will produce a solution on an operational time scale, even for a very large system of equations. The convergence of iterative solvers can be further enhanced by effective choice of preconditioner. If we can improve the condition number of the matrix involved in the solve, we reduce the number of iterations required to converge to a solution. We will see an example of a preconditioned method in algorithm 5.1, where we modify the problem we solve. Instead of solving $A\underline{x} = \underline{b}$ we instead solve $P^{-1}A\underline{x} = P^{-1}\underline{b}$, where P is our preconditioner.

We want to chose P such that it is easy to invert and $P^{-1}A$ has more of the desirable properties that will speed up convergence. In section 5.1.3, we will see that the convergence of the conjugate gradient method is dependent on the condition number of A. If we precondition, the convergence is now dependent on the condition number $\kappa(P^{-1/2}AP^{-1/2})$.

Preconditioners usually lie somewhere between two extremes, $P = I$ and $P = A^{-1}$. With the trivial preconditioner, $P = I$, we do not change the system at all and we must solve the original problem. The other extreme, $P = A^{-1}$, is as much work as solving the original problem equation (5.1). The art of preconditioning is finding some matrix between these two extremes, which is not too difficult to calculate, but improves the condition number $\kappa(P^{-1/2}AP^{-1/2})$ sufficiently that the iterative method converges faster.

In sections 5.2 and 5.3 we will look at other solvers that exploit the structure of the underlying matrix to solve the system more effectively. By combining these techniques with different solvers, we will construct a range of viable solvers and preconditioners.

5.2.2 Approximate Schur complement for a DG problem

At the end of chapter 3, we obtained a matrix equation, which when solved yields a numerical solution to the shallow water equations. We previously had the matrix $\underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\mathrm{L}}}$ that was used for implicit timestepping. This matrix comes from equation (3.45) and can be written

$$\underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\mathrm{L}}} = \left(\begin{array}{c|c|c} M + \Delta t Q_{\phi\phi} & \Delta t(-D_x + Q_{\phi U_1}) & \Delta t(-D_y + Q_{\phi U_2}) \\ \hline \Delta t(-D_{B,x} + Q_{U_1\phi} - M_{B,x}) & M + \Delta t Q_{U_1U_1} & \Delta t(Q_{U_1U_2} - fI) \\ \hline \Delta t(-D_{B,y} + Q_{U_2\phi} - M_{B,y}) & \Delta t(Q_{U_2U_1} + fI) & M + \Delta t Q_{U_2U_2} \end{array}\right). \tag{5.8}$$

We already know this matrix has block structure, which is shown in section 3.3.3. Recall that $M, D_x, D_y, D_{B,x}, D_{B,y}$ and I are all block diagonal, it is only the matrices Q, corresponding to the numerical flux that couple the degrees of freedom and appear as off block diagonal terms.

We will now construct a preconditioner by using an approximate Schur complement factorisation. This allows us to approximate the inverse of the matrix $\underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\mathrm{L}}}$. To ensure the Schur complement is not dense we make a diagonal approximation to the bottom right block and form the Schur complement in the top left.

To assess its effectiveness we want to perform some analysis on this preconditioner, to do so we make some simplifications. We will consider a simplified shallow water equation in one dimension and discretise using a degree zero DG method.

Example 5.2.1 (A preconditioner for the simplified SWE)**.** We will consider the same problem, but in one spatial dimension and with periodic boundary conditions, so the U_2 component and D_y terms disappear. It no longer makes sense to include a Coriolis term in

one dimension, so $f = 0$. The bathymetry can be made constant, which makes $M_{B,x} = 0$ and $D_{B,x} = D_x$.

This leaves us with the modified matrix A

$$A = \left(\begin{array}{c|c} M+Q & -D_x+R \\ \hline D_x-R & -(M+Q) \end{array}\right) \in \mathbb{R}^{2n\times 2n}. \tag{5.9}$$

Finally, we use a lowest order (degree zero) DG method, which means the mass matrix $M = hI$, $D_x = 0$. The Δt is now absorbed into the definition of Q and R, and we define a new parameter μ based on selecting a suitable timestep size. From chapter 4 section 4.2, we know that the timestep must satisfy the CFL condition for the timestepping regime to remain stable. Hence we define the parameter

$$\mu := 2\Delta t \leq \frac{C_{\text{CFL}}\Delta x}{c_g} = hC_{\text{CFL}} \tag{5.10}$$

With this new parameter we define

$$Q := \frac{h^2\mu}{2} \underbrace{\left[\frac{1}{h^2}\begin{pmatrix} 2 & -1 & 0 & \cdots & 0 & -1 \\ -1 & 2 & -1 & & & 0 \\ 0 & -1 & 2 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & & \\ 0 & & & & 0 & 1 \\ -1 & 0 & \cdots & & -1 & 2 \end{pmatrix}\right]}_{J}$$

and

$$R := \frac{h\mu}{2} \underbrace{\left[\frac{1}{h}\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & -1 \\ -1 & 0 & 1 & & & 0 \\ 0 & -1 & 0 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & & \\ 0 & & & & 0 & 1 \\ 1 & 0 & \cdots & & -1 & 0 \end{pmatrix}\right]}_{K}.$$

Note 5.2.2. Here the highlighted part of Q corresponds to a discrete diffusion operator. That is $J \to \partial_x^2$ as $h \to 0$, but we have to take a factor of h^2 out to see this clearly. The highlighted part of R is a discrete differentiation operator ($K \to \partial_x$ as $h \to 0$). Since $R^\top = -R$, we have multiplied the bottom blocks by -1 to make the matrix symmetric.

To build a preconditioner we can approximate the bottom right block of the matrix in equation (5.9) with the diagonal matrix

$$\widetilde{M} = h\rho I, \tag{5.11}$$

where ρ is a parameter that we can choose. This means that our preconditioner matrix looks like

$$\widetilde{A} = \begin{pmatrix} M+Q & R \\ R^\top & -\widetilde{M} \end{pmatrix} \in \mathbb{R}^{2n\times 2n}.$$

To assess how this will affect our solver we must look at the eigenvalues of $\widetilde{A}^{-1}A$. To do this we prove the original result in proposition 5.2.3 showing that the lowest order

preconditioned equation has real, positive and bounded eigenvalues.

Proposition 5.2.3 (Real and positive eigenvalues)**.** *For $\rho > 0$, the eigenvalues of $\widetilde{A}^{-1}A$ are real, positive and can be bounded above and below by functions of ρ. That is, for any λ_A an eigenvalue of A, we have $\lambda_A \in \mathbb{R}$ and $\lambda_A = 1$ or*

$$0 < \frac{\sigma}{\rho(\sigma+2)} \leq \lambda_A \leq \frac{1+2\sigma^{-1}}{\rho},$$

where $\sigma := \frac{h}{\mu}$. The matrices A and $\widetilde{A}$ are defined in example 5.2.1.

Proof. The eigenvalues of $\widetilde{A}^{-1}A$ are λ_A, where

$$\widetilde{A}^{-1}A\underline{q} = \lambda_A \underline{q}.$$

Which is equivalent to looking at the generalised eigenvalue problem

$$A\underline{q} = \lambda_A \widetilde{A}\underline{q}.$$

To find these eigenvalues, we use Fourier methods. Working on the periodic domain $[-\pi, \pi]$ we have the Fourier basis

$$e_k^j = \frac{1}{\sqrt{2\pi}} e^{ikx_j} = \frac{1}{\sqrt{2\pi}} e^{ik(-\pi+jh)}, \text{ for } j = 0, \ldots, n-1, \ k = 0, 1, \ldots, n-1,$$

where $x_j = -\pi + jh$ and $h = 2\pi/n$. x_n is identified with x_0, due to the periodic boundary conditions. Note i here is the unit imaginary number and not an index. Further, note

$$e_k^{j\pm1} = \frac{1}{\sqrt{2\pi}} e^{ik(-\pi+jh\pm h)} = \frac{1}{\sqrt{2\pi}} e^{ik(-\pi+jh)} e^{\pm ikh} = e_k^j e_k^{\pm ikh}.$$

We make and ansatz that an eigenvector for M, Q, R and $\widetilde{M}$ has the form

$$\underline{e}_k = (e_k^0, e_k^1, \ldots, e_k^{N-1})^\top.$$

We start by considering the eigenvalues of the $M + Q$ matrix

$$(M+Q)\underline{e}_k = \lambda_Q(h,k)\underline{e}_k,$$

where

$$\lambda_Q(h,k) = \left(-\frac{\mu}{2}e_k^{-ikh} + (h+\mu) - \frac{\mu}{2}e_k^{ikh}\right) = (h + \mu + \mu\cos(hk)).$$

For the matrix R we have

$$R\underline{e}_k = \lambda_R(h,k)\underline{e}_k,$$

where

$$\lambda_R(h,k) = \frac{\mu}{2}\left(e_k^{ikh} - e_k^{-ikh}\right) = \mu i \sin(hk)$$

For $\widetilde{M}$ our approximation to the bottom right block, we simply have

$$\widetilde{M}\underline{e}_k = h\rho\underline{e}_k$$

since $\widetilde{M}$ is diagonal.

Now we can look at A and $\widetilde{A}$ acting on q_k and make the ansatz:

$$\underline{q}_k = \begin{pmatrix} \phi_k \underline{e}_k \\ U_k \underline{e}_k \end{pmatrix},$$

where $\phi_k, U_k \in \mathbb{C}$.

We define the two matrices F and $\widetilde{F}$ using the action of A and $\widetilde{A}$.

$$\begin{aligned} A\underline{q}_k &= \begin{pmatrix} M+Q & R \\ R^\top & -(M+Q) \end{pmatrix} \begin{pmatrix} \phi_k \underline{e}_k \\ U_k \underline{e}_k \end{pmatrix} \\ &= \underbrace{\begin{pmatrix} \lambda_Q & \lambda_R \\ -\lambda_R & -\lambda_Q \end{pmatrix}}_{=:F} \begin{pmatrix} (\phi_k \underline{e}_k) \\ (U_k \underline{e}_k) \end{pmatrix}, \end{aligned} \qquad \begin{aligned} \widetilde{A}\underline{q}_k &= \begin{pmatrix} M+Q & R \\ R^\top & -\widetilde{M} \end{pmatrix} \begin{pmatrix} \phi_k \underline{e}_k \\ U_k \underline{e}_k \end{pmatrix} \\ &= \underbrace{\begin{pmatrix} \lambda_Q & \lambda_R \\ -\lambda_R & -h\rho \end{pmatrix}}_{=:\widetilde{F}} \begin{pmatrix} (\phi_k \underline{e}_k) \\ (U_k \underline{e}_k) \end{pmatrix}. \end{aligned}$$

To find the eigenvalues λ_A, we look at $\det(F - \lambda_A \widetilde{F}) = 0$, that is

$$\begin{aligned} \det(F - \lambda_A \widetilde{F}) =& \left| \begin{pmatrix} \lambda_Q(1-\lambda_A) & \lambda_R(1-\lambda_A) \\ -\lambda_R(1-\lambda_A) & -\lambda_Q + \lambda_A h\rho \end{pmatrix} \right| \\ =& -\lambda_Q(1-\lambda_A)(\lambda_Q - \lambda_A h\rho) + \lambda_R^2(1-\lambda_A)^2 \\ =& (1-\lambda_A)\left[\lambda_R^2(1-\lambda_A) - \lambda_Q(\lambda_Q - \lambda_A h\rho)\right] \\ =& (1-\lambda_A)\left[\lambda_R^2 - \lambda_Q^2 - \lambda_A(\lambda_R^2 - \lambda_Q h\rho)\right] \\ =& 0. \end{aligned}$$

We find that either $\lambda_A = 1$, or substituting the expressions for the eigenvalues λ_M, λ_Q and λ_R. For brevity we define $c := \cos(hk)$ and $s := \sin(hk)$.

$$\begin{aligned} \lambda_A &= \frac{\lambda_R^2 - \lambda_Q^2}{\lambda_R^2 - h\rho\lambda_Q} \\ &= \frac{-\mu^2 s^2 - (h + \mu(1-c))^2}{-\mu^2 s^2 - h\rho(h + \mu(1-c))} \\ &= \frac{s^2 + \left(\frac{h}{\mu} + (1-c)\right)^2}{s^2 + \frac{h}{\mu}\rho\left(\frac{h}{\mu} + (1-c)\right)} \end{aligned}$$

This means λ_A is always both real and positive.

To establish an upper bound for the eigenvalues, we first define the constants $\sigma := \frac{h}{\mu}$ and

$$\begin{aligned} C &:= \max\left\{1,\ \frac{1}{\rho}(1 + 2\sigma^{-1})\right\} \\ &\geq \frac{1}{\rho}\left[1 + \sigma^{-1}(1-c)\right] \end{aligned}$$

which implies that $\sigma + 1 - c \leq C\sigma\rho$, then

$$\lambda_A \leq \frac{Cs^2 + C\sigma\rho(\sigma + 1 - c)}{s^2 + \sigma\rho(\sigma + 1 - c)} = C.$$

To establish a lower bound for the eigenvalues we use the identity

$$\frac{z+a}{z+b} - \frac{a}{b} \geq 0 \quad \text{for } z \geq 0, \quad b > a > 0.$$

Since $\sigma \leq \sigma + 1 - c \leq \sigma + 2$ we can bound λ_A from below:

$$\lambda_A \geq \frac{s^2 + \sigma^2}{s^2 + \sigma\rho(\sigma+2)} \geq \frac{\sigma}{\rho(\sigma+2)}$$

Hence for any λ_A an eigenvalue of A, we have $\lambda_A \in \mathbb{R}$ and $\lambda_A = 1$ or

$$0 < \frac{\sigma}{\rho(\sigma+2)} \leq \lambda_A \leq \frac{1 + 2\sigma^{-1}}{\rho}.$$

This gives the bounds in proposition 5.2.3. QED

Since σ^{-1} is directly proportional to the CFL number, we can get a rough idea of what the bounds are. The CFL number is approximately 10 for an implicit timestepping scheme, which means that

$$\frac{1}{21\rho} \leq \lambda_A \leq \frac{21}{\rho}.$$

The diagonal of the matrix $M + Q$ is $h + \mu$ and so $\rho \approx 1$ might be a typical choice, although we are free to choose ρ as we please. We see by calculating typical bounds that the eigenvalues can be far from 1 and arbitrarily close to 0, depending on the choice of ρ.

5.2.3 Numerical experiments

The bounds established in proposition 5.2.3 only hold for a lowest order (degree zero) DG method, but we can numerically calculate the spectra for higher order methods, to see if the eigenvalues are also bounded. Figure 5-1 shows numerically the calculation we have just performed in the proof of proposition 5.2.3. We start with a coarse mesh where $h = 1/4$ and pick μ so that $\sigma^{-1} = 10$ and the CFL condition is satisfied. For the approximation $\widetilde{M}$ take equation (5.11) with $\rho = 1$.

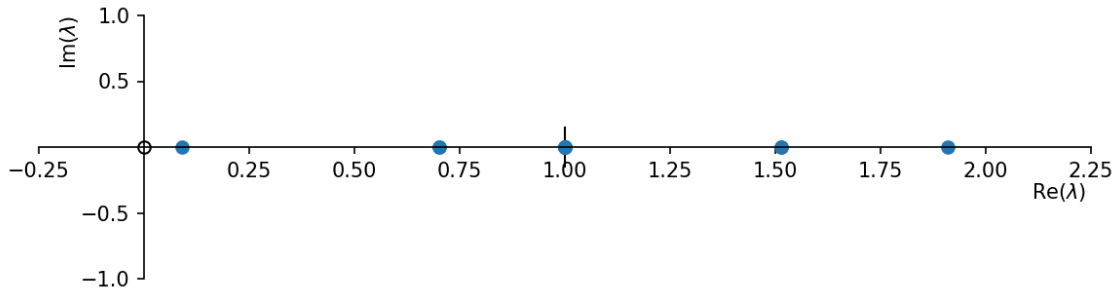

Figure 5-1: Spectrum for $\widetilde{A}^{-1}A$, where A uses lowest order DG

The blue dots in figure 5-1 represent the eigenvalues of A in the complex plane when using a degree zero DG method. As proven above, they lie on the real line, are strictly greater than 0 and $1/21 \leq \lambda_A \leq 21$. Notice that the largest eigenvalue is just less than 2, well below 21, the upper bound predicted by proposition 5.2.3. The smallest eigenvalue is much closer to the lower bound of $1/21 \approx 0.05$. Since the condition number for SPD matrices is equal to the ratio of the largest and smallest eigenvalues in absolute value

$(\max_n |\lambda_n|)$, we know that $\kappa(\widetilde{A}^{-1}A) \approx 40$.

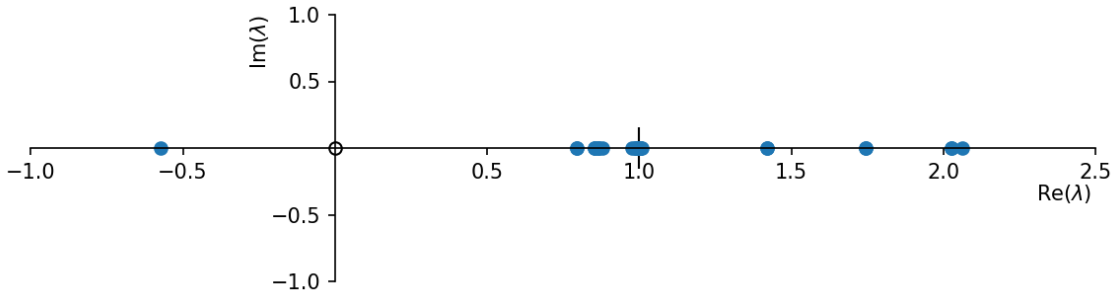

Figure 5-2: Spectrum for $\widetilde{A}^{-1}A$, where A that uses order 4 DG

Figure 5-2 shows what happens when we consider a higher-order DG method, in this case a degree 4 polynomial approximation is used on each cell. The h remains fixed, but to keep the CFL number equal to 10, we must reduce the parameter μ, by using the definition of μ in equation (5.10). The spectrum for $\widetilde{A}^{-1}A$ still lies on the real axis, and there is a large cluster of eigenvalues around 1, however some eigenvalues have now crossed the imaginary axis and lie in the left half plane. This means that this matrix is no longer positive definite.

Despite this, the condition number is better than the lowest order case. Using figure 5-2 we estimate that the degree 4 case has condition number $\kappa(\widetilde{A}^{-1}A) \approx 4.5$.

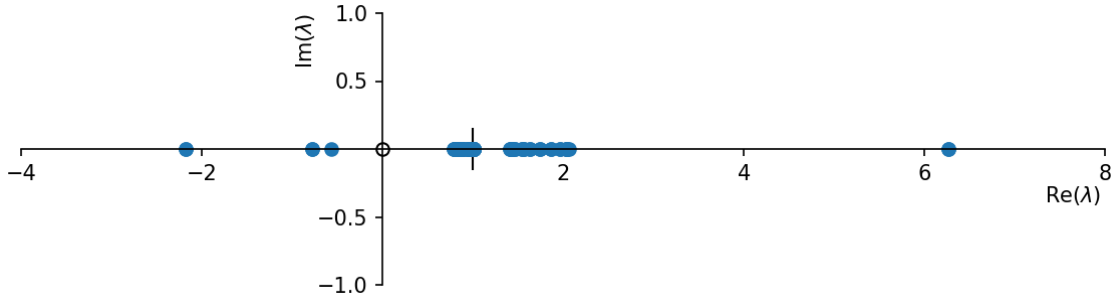

Figure 5-3: Spectrum for $\widetilde{A}^{-1}A$, where A uses order 4 DG and smaller grid

Figure 5-3 demonstrates that the matrix remains indefinite as we refine the mesh we work on. Refining the mesh makes h smaller and also increases the size of the matrix A. It also has the effect of moving the negative eigenvalues of our preconditioned matrix further away from zero.

From figure 5-3, we estimate $\kappa(\widetilde{A}^{-1}A) \approx 13$, so the preconditioned matrix condition number is still increasing as we refine the mesh. This means the condition number of the matrix $\widetilde{A}^{-1}A$ is larger and the number of iterations the solver takes to converge increases. Since the whole point of preconditioning was to speed up the rate of convergence by reducing the number of iterations, we would like to keep κ small.

While the numerical experiments for higher orders show that the eigenvalues are not positive and bounded away from zero, what we have demonstrated is that this preconditioner *is* effective for lowest order DG methods. The proof of proposition 5.2.3 guarantees that what we saw in figure 5-1, that the eigenvalues are real and positive valued, holds for all lowest order DG matrices of the form in example 5.2.1. The next section outlines an alternative approach.

5.3 Hybridisation as a factorisation

In the previous section we saw that an approximate Schur complement is a good preconditioner for the DG discretisation of the SWE at lowest order, but for higher orders this is no longer guaranteed to be the case. Since we are primarily interested in higher order methods we must take an alternative approach.

For our approximate Schur complement preconditioner, we removed off (block) diagonal entries in an effort to make the global matrix easier to invert. The spectral analysis in the previous section suggests that these off diagonal entries are too important to ignore at higher order as they change the properties of the matrix significantly. We saw in chapter 4 that we can use hybridisation replace coupled degrees of freedom between cells and form an equivalent problem utilising additional variables on the mesh facets. This increases the size of the global matrix, but also changes the block structure.

We saw in equation (3.47) that we can write the matrix system for the SWE as

$$(\underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\mathrm{L}}})\underline{q} = \underline{b}. \tag{5.12}$$

Recall that $\underline{\underline{\mathrm{M}}}$ has only entries on the block diagonal, but $\underline{\underline{\mathrm{L}}}$ contains off diagonal entries. The vector $\underline{b}$ represents the right-hand side of the timestepping method, and it's exact value is not important for this discussion.

The size of the system matrices here depends on the mesh size. If $\underline{\underline{\mathrm{M}}}, \underline{\underline{\mathrm{L}}} \in \mathbb{R}^{m \times m}$, then $m = N(d+1)\gamma(p+1)^d$, where N is the number of cells in the mesh and $\gamma(p+1)^d$ is the number of degrees of freedom that were chosen to represent a function on each cell. Here p is the polynomial degree of our basis functions and d is the geometric dimension of the domain we solve the underlying PDE in, for SWE $d = 2$. For a mesh of quadrilaterals $\gamma = 1$ and for a mesh of triangles $\gamma = \frac{p+2}{2(p+1)}$.

When we hybridise the SWE, as in section 4.1, the numerical flux couples to a trace variable, which is denoted $\underline{\lambda}$. This is instead of the numerical flux coupling one cell to its neighbour in the DG method, which is the source of off block diagonal terms.

Hybridisation leads to a block matrix problem, shown in equation (4.10),

$$\left(\begin{array}{c|c} \underline{\underline{\mathrm{M}}} + \Delta t \widehat{\underline{\underline{\mathrm{L}}}} & \Delta t \underline{\underline{\mathrm{G}}} \\ \hline \Delta t \underline{\underline{\mathrm{G}}}^{\top} & \Delta t \underline{\underline{\mathrm{T}}} \end{array}\right) \begin{pmatrix} \underline{q} \\ \underline{\lambda} \end{pmatrix} = \begin{pmatrix} \underline{b} \\ \underline{0} \end{pmatrix}. \tag{5.13}$$

The blocks that make up $\widehat{\underline{\underline{\mathrm{L}}}}$ are now block diagonal, just like $\underline{\underline{\mathrm{M}}}$, giving $\underline{\underline{\mathrm{M}}} + \Delta t \widehat{\underline{\underline{\mathrm{L}}}}$ a structure that makes it easier to invert than the matrix in equation (5.12).

Since the top left block is now easier to invert, it allows us to form the Schur complement in the bottom right block. And because $\underline{\underline{\mathrm{M}}} + \Delta t \widehat{\underline{\underline{\mathrm{L}}}}$ now has block diagonal structure, this ensures that the Schur complement $\Delta t \underline{\underline{\mathrm{T}}} - (\Delta t)^2 \underline{\underline{\mathrm{G}}}^{\top} (\underline{\underline{\mathrm{M}}} + \Delta t \widehat{\underline{\underline{\mathrm{L}}}})^{-1} \underline{\underline{\mathrm{G}}}$ is a sparse matrix. We can use a suitable iterative solver[3] to solve the bottom right block system. Since the Schur complement factorisation is an exact factorisation, we do not require an outer solver, and the solution can be reconstructed from the solution to the bottom right block system.

$\underline{\underline{\mathrm{M}}} + \Delta t \widehat{\underline{\underline{\mathrm{L}}}}$ is the same size as $(\underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\mathrm{L}}}) \in \mathbb{R}^{m \times m}$. Since $\underline{\underline{\mathrm{T}}}$ acts on the trace degrees of freedom, its size depends on the number of interior facets in the mesh. If $\underline{\underline{\mathrm{T}}} \in \mathbb{R}^{n \times n}$, then $n = \omega N(d+1)(p+1)^{d-1}$, where N, p and d are the number of cells, polynomial degree and dimension as discussed above. For a mesh of triangles $\omega = 3$ and for quadrilaterals $\omega = 4$, where we have assumed all facets are internal facets (which is true if we have a periodic

[3] CG if the matrix is SPD, otherwise GMRES

domain).

The value of n can be further reduced by using properties of the underlying PDE, as we did in section 4.1.1 where we eliminated some of the trace variables. Importantly, n is much smaller than m. For the SWE, without eliminating any of the trace variables, n is $(p+2)/6$ times smaller than m for a mesh triangles and $(p+1)/3$ times smaller for a mesh of quadrilaterals.

Now we can use algorithm 5.2, the algorithm which exploits the Schur factorisation, to solve the new block system in equation (5.13). Since $(\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}})$ is the easier block to invert, and since the $\underline{\underline{\mathrm{T}}}$ block is smaller, it will be more effective to form the Schur complement in the bottom right block. Rewriting equation (5.7) as

$$\begin{pmatrix} H & G \\ F & E \end{pmatrix} \begin{pmatrix} \underline{z} \\ \underline{y} \end{pmatrix} = \begin{pmatrix} \underline{g} \\ \underline{f} \end{pmatrix}$$

we see the two methods are equivalent.

In the first step of the Schur complement factorisation algorithm, we calculate a modified right-hand side by first solving

$$(\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}})\underline{r} = \underline{b}. \tag{5.14}$$

Refer back to equation (5.13) for each variable. This solve is inexpensive as $\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}}$ has block structure. If the Schur complement factorisation of $\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}}$ is formed all of the matrix blocks are themselves block diagonal.

The modified right-hand side is $-\Delta t\underline{\underline{\mathrm{G}}}^{\top}\underline{r}$ and the Schur complement of the matrix in equation (5.13) is $\Delta t\underline{\underline{\mathrm{T}}} - (\Delta t)^2\underline{\underline{\mathrm{G}}}^{\top}(\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}})^{-1}\underline{\underline{\mathrm{G}}}$.

The next solve in algorithm 5.2 is the system

$$\left(\Delta t\underline{\underline{\mathrm{T}}} - (\Delta t)^2\underline{\underline{\mathrm{G}}}^{\top}(\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}})^{-1}\underline{\underline{\mathrm{G}}}\right)\underline{\lambda} = -\Delta t\underline{\underline{\mathrm{G}}}^{\top}\underline{r}. \tag{5.15}$$

This is the more expensive solve and we will use a powerful solver like GMRES or multigrid on this system to solve the whole system efficiently.

The final step is to recover the solution $\underline{q}$ from the trace variable $\underline{\lambda}$. This is done by solving

$$(\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}})\underline{q} = \underline{b} - \Delta t\underline{\underline{\mathrm{G}}}\,\underline{\lambda}. \tag{5.16}$$

This solve is also inexpensive, for the same reasons the first solve in equation (5.14) was. The factorisation can even be saved from equation (5.14) so that it does not have to be recomputed for the solve in equation (5.16). The procedure that our implementation uses is outlined in section 7.3.

In the next section we look at multigrid, which is an efficient solver for the trace solve in equation (5.15).

5.4 Multigrid

Multigrid is a very popular solver (and also preconditioning technique) for elliptic PDEs due to its ability to reduce the error at all length scales [14, 54]. It has $O(n)$ complexity for an equation with n degrees of freedom when using full multigrid. There are two distinct multigrid techniques: Algebraic multigrid and geometric multigrid.

Geometric Multi-Grid (GMG) uses the structure of the mesh used in the discretisation. To take advantage of this, we are required to define a grid hierarchy. Furthermore, we

also need to specify how to move the residual from one grid to the next in the hierarchy and how to move back again. This can be as simple as subdividing the cells in the underlying mesh, to construct a simple mesh hierarchy, and using the simple injection for the prolongation operator. Although originally developed for finite difference and finite volume discretisations, multigrid can be applied to DG methods by formulating it with bilinear forms (see section 6.1). This requires defining operators that move a residual function from a function space on one mesh to a new function space on a different mesh.

Algebraic Multi-Grid (AMG), on the other hand, looks at the structure of the system matrix only. One variant of AMG is aggregation based AMG [52]. This involves constructing a graph based on the structure of the matrix in a manner similar to the way a graph can be produced from an adjacency matrix. This graph is then coarsened by combining groups of vertices to produce a hierarchy of nested graphs allowing the multigrid algorithm to be performed. The coarse space matrices can automatically be deduced from the coarser graphs, as can the grid transfer operators.

Since AMG only requires an assembled matrix, we can immediately use this tool as a solver. However, AMG has a high set up cost, due to the coarse matrix construction method. It is also an entirely matrix based method, so there is no matrix free option if AMG is used. If we want to use GMG we must think about what mesh and function space we want to construct our mesh hierarchy on.

For the shallow water equations we want to combine GMG with a hybridised method, but there is an additional difficulty: Since the hybridised system uses degrees of freedom that live on the facets of the mesh (as described in section 4.1), constructing a mesh hierarchy is more difficult. We would like to move from the facet DOFs back to cell DOFs, where we already know how to perform multigrid. When we consider this issue of moving from the facets back to cells in terms of function spaces we see that the these are not nested. So to perform GMG on the facets of the mesh we need to define transfer operators between two non-nested function spaces. This process is described in section 6.3.

5.4.1 GMG Overview

With a specified mesh hierarchy, there are three key steps to the GMG algorithm:

- **Smooth** is used to reduce the high frequency components of the error at a given grid level. This is typically a few iterations of a relaxation method, such as Jacobi or SOR.
- **Restriction** for moving a problem from a finer grid to a coarser one.
- **Prolongation** for moving a problem from a coarser grid to a finer one.

With these three steps we can demonstrate the geometric multigrid method, which we do in the following example.

Example 5.4.1 (Modified Helmholtz)**.** We consider the modified Helmholtz problem in 1D:

$$-\partial_x^2\phi + k^2\phi = f \quad \text{on} \quad \Omega = [0,1] \tag{5.17}$$

The Dirichlet boundary conditions are $\phi(0) = 0$, $\phi(1) = 1$. If the right-hand side is

$$f(x) = 2 + k^2(1-x^2) - (k^2\ell^2\pi^2)\cos(\ell\pi x),$$

where ℓ is an odd integer (for this example take $\ell = 3$), then

$$\phi_{\text{exact}}(x) = 1 - x^2 - \cos(\ell\pi x)$$

is an exact solution that we can compare our numerical solution to.

To demonstrate the multigrid method we now find a solution using a finite difference discretisation. We start by constructing a finite difference scheme on the finest mesh, which for this example will use $N+1$ points, where $N = 2^n$. This gives a grid spacing of $h = 1/N$. We chose points $x_i = ih$ for $i = 1, \ldots, N-1$ on the interior of the domain.

The boundary conditions are satisfied by fixing $\phi_0 = 0$ and $\phi_N = 1$ and we only numerically solve for the points on the *interior* of the domain. The right-hand side is

$$\underline{f}_h = \left(f(x_1) - \frac{\phi_0}{h^2},\ f(x_2),\ \ldots,\ f(x_{N-2}),\ f(x_{N-1}) - \frac{\phi_N}{h^2} \right)^\top \in \mathbb{R}^{N-1},$$

the matrix form of the differential operator operator $\partial_x^2 + k^2$ for finite differences is

$$A_h := \frac{1}{h^2} \begin{pmatrix} 2+k^2h^2 & -1 & 0 & 0 & \cdots \\ -1 & 2+k^2h^2 & -1 & 0 & \\ 0 & -1 & 2+k^2h^2 & -1 & \\ \vdots & & & \ddots & \end{pmatrix} \in \mathbb{R}^{(N-1)\times(N-1)} \tag{5.18}$$

and the solution vector is $\underline{\phi}_h := (\phi_1, \ldots, \phi_{N-1})^\top \in \mathbb{R}^{N-1}$, notice that $\underline{f}_h, A_h$ and $\underline{\phi}_h$ all depend on the grid spacing h.

We now demonstrate how to solve the matrix equation

$$A_h \underline{\phi}_h = \underline{f}_h$$

using the multigrid method. We start with an initial guess $\left[\underline{\phi}_h\right]_i = ih$ for $i = 1, \ldots, N-1$, which is just a linear interpolation between the fixed boundary conditions. To construct a mesh hierarchy, we will double the grid spacing between levels to obtain the next coarsest mesh. By doing this, the coarser meshes are naturally nested within the finer meshes. That is, every other fine grid point corresponds to a coarse grid point.

Evaluating the analytic solution at the points on the interior of the domain we get $\left[\underline{\phi}_{\text{exact}}\right]_i := \phi_{\text{exact}}(x_i)$. This allows us to define the numerical error on the finest level as $\underline{e}_h = \underline{\phi}_{\text{exact}} - \underline{\phi}_h$.

Figure 5-4 shows the numerical error at *each* level of our 3–level example multigrid scheme where each of the plotted lines corresponds to a different number of smoothing interations. To make this example concrete, we chose $k = 1, \ell = 3$ and $n = 4$ on the finest mesh, so the grid spacing is $h = 1/16$. The grid spacing is doubled on each new level, so the coarsest mesh has grid spacing $4h = 1/4$. The top left plot in the box in figure 5-4 (labelled ⓐ) shows the initial guess as a blue line and the exact solution as a red dotted line.

The first step of the multigrid method is to smooth the equation on the finest mesh

$$A_h \underline{\phi}_h = \underline{f}_h.$$

In our example we will use 0, 1 or 2 iterations of an SOR method weighted by $\omega = 0.6$ for our smoother.

In figure 5-4 the blue lines corresponds to no smoothing, the orange lines to one iteration of the SOR smoother, and the green lines to two iterations of the SOR smoother. The second plot in the left-hand column of figure 5-4 (labelled ⓑ) shows the difference between the analytic solution and the computed solution on a grid with 15 interior points.

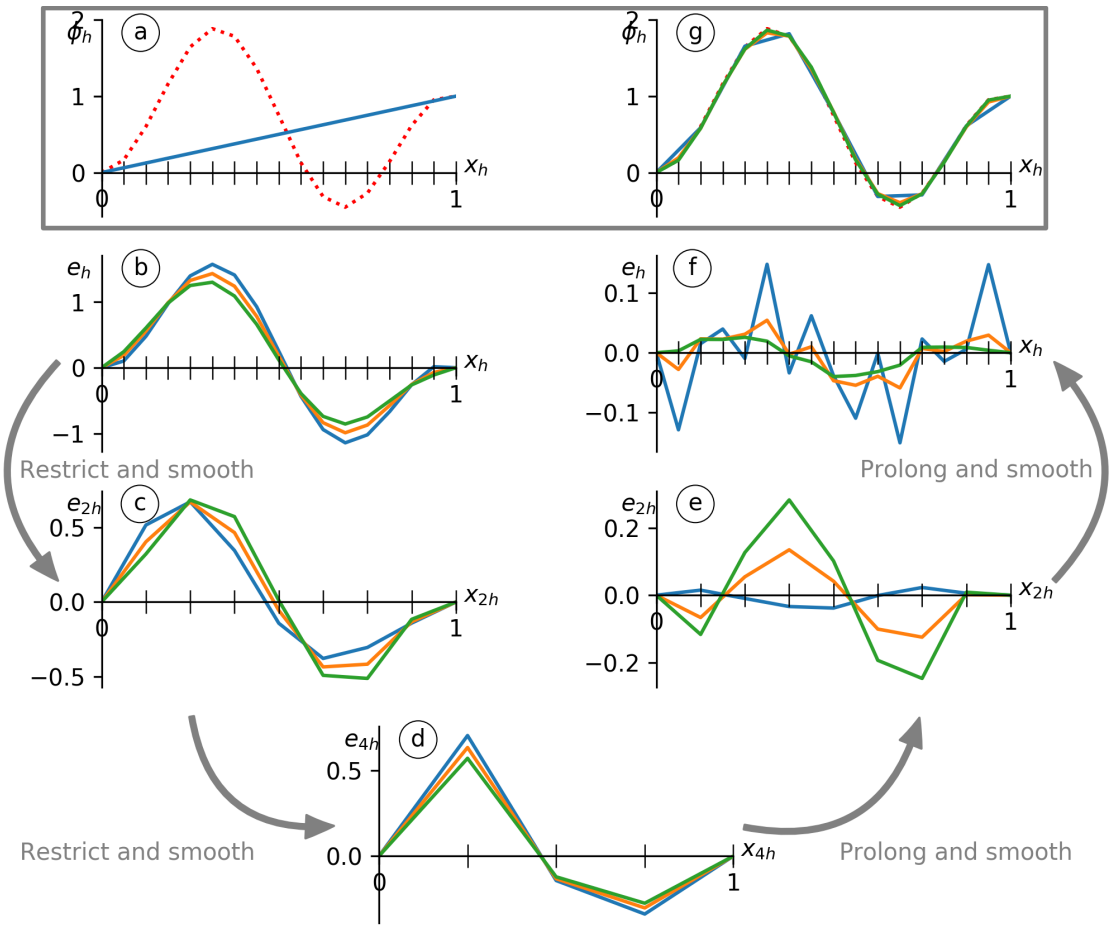

Figure 5-4: Graphical representation of the error propagation in a multigrid V-cycle. Blue lines correspond to the error with no smoothing, orange with one smoothing iteration per level and green with two smoothing steps per level. A full description of each plot is included in the text of example 5.4.1.

After smoothing we calculate the residual at the finest level

$$\underline{r}_h = \underline{f}_h - A_h \underline{\phi}_h.$$

This residual can be transferred to a coarser mesh via the restriction operation. The coarser mesh has grid spacing $2h$, hence uses points $x_i = 2ih$. For the restriction we could simply takes every other entry in the vector $\underline{r}_h \in \mathbb{R}^{15}$ to get the new vector $\underline{r}_{2h} \in \mathbb{R}^7$, so that $[\underline{r}_{2h}]_i = [\underline{r}_h]_{2i}$. Instead we use the full weighting operator, so that

$$[\underline{r}_{2h}]_i = \frac{1}{4}([\underline{r}_h]_{2i-1} + 2[\underline{r}_h]_{2i} + [\underline{r}_h]_{2i+1}).$$

The full weighting operator is useful as, in this case, it is the restriction *induced by the prolongation*, this property is discussed in more detail for a function based formulation in section 6.1. So we have $R(\underline{r}_h) = \underline{r}_{2h}$, where R is the restriction operator.

Since the matrix $A_h \in \mathbb{R}^{15\times 15}$ was defined using the grid spacing h, we can easily obtain the matrix on the coarser space, which is just $A_{2h} \in \mathbb{R}^{7\times 7}$. A_{2h} is just A_h in equation (5.18), but with h replaced with $2h$ everywhere. We can then find the approximate error on the coarser space by smoothing the equation

$$A_{2h}\underline{\phi}_{2h} = \underline{r}_{2h}$$

using an initial guess $\underline{\phi}_{2h} = \underline{0}$. The third plot in the left hand column of figure 5-4 (labelled ⓒ) shows the error on a grid with 7 interior points.

We can find the residual on the grid with spacing $2h$ and repeat the restriction a second time to get a residual $\underline{r}_{4h}$ on the coarsest space. We can also compute the matrix A_{4h}. In the concrete example shown in figure 5-4 there are only have 3 interior points on the coarsest level (bottom plot, labelled ⓓ). At this stage the matrix is small enough that we can use a direct solve to find the approximate error $\underline{\phi}_{4h}$, rather than using a smoother. The line plotted in the bottom plot of figure 5-4 (labelled ⓓ) is the approximate error on a grid with 3 interior points.

Now that we know the error on the coarsest level we want to propagate this back up to the finest level. This is done using prolongation, which for this 1D mesh is piecewise linear interpolation[4]. Specifically, the value of the error at the point x_{2i+1} on the fine space, which is situated halfway between the points x_i and x_{i+1} on the coarse mesh, is just $\frac{1}{2}(e_i + e_{i+1})$.

The correction is denoted $\underline{e}_{2h} = P(\underline{\phi}_{4h})$, where P is the prolongation operator. We update $\underline{\phi}_{2h}$ by changing its value to

$$\underline{\phi}_{2h} = \underline{\phi}_{2h} + P(\underline{\phi}_{4h}) = \underline{\phi}_{2h} + \underline{e}_{2h},$$

then apply the smoother again on the grid with spacing $2h$. The numerical error on the grid with 7 interior points (after it has been smoothed) is shown in the third plot from the top in the right-hand column of figure 5-4 (labelled ⓔ).

This correction is then prolongated back to the finest grid to get the correction $\underline{e}_h$. By updating ϕ_h to

$$\underline{\phi}_h = \underline{\phi}_h + P(\underline{\phi}_{2h}) = \underline{\phi}_h + \underline{e}_h,$$

the final solution is found after applying a few more smoothing iterations.

The plot second from the top in the right hand column in figure 5-4 (labelled ⓕ)

[4] again a more sophisticated interpolation could be employed

shows the (smoothed) numerical error on the finest grid with 15 interior points. Notice that in this plot the effect of smoothing is very clear. The blue line corresponding to no smoothing shows that the high frequency error, which is created by prolongation, is still present, whereas the green line corresponding to only 2 iterations of the SOR smoother is a smoother line and much closer to 0.

The top right plot of figure 5-4 (labelled ⓖ) shows the solution on the finest mesh, which very closely matches the analytic solution plotted as a dotted red line.

Example 5.4.1 showed how it was possible to perform geometric multigrid in a simple 1D finite difference case. In chapter 6 we discuss the multigrid method for a DG discretisation.

We will now write down the multigrid method abstractly for a general matrix problem. The problem we wish to solve is $A\underline{x} = \underline{b}$, where $\underline{x}$ is the true solution. At the finest level we will label the matrix A_h and denote the approximate solution $\underline{x}_h$. We take the problem $A_h\underline{x}_h = \underline{b}_h$ defined on the finest mesh and smooth it. We then calculate the residual $\underline{r}_h := \underline{b}_h - A_h\underline{x}_h$ and restrict it to a new problem $A_{2h}\underline{e}_{2h} = \underline{r}_{2h}$ on a coarser mesh. On the coarser mesh we solve for the error $\underline{e}_{2h}$, which is prolongated back to the fine mesh to correct the approximate solution $\underline{x}_h$ and smoothed again.

For algorithm 5.3 we have denoted the matrix A, solution vector $\underline{x}$, the error $\underline{e}$ and the residual $\underline{r}$ on the finest level with a subscript h. We have used a coarser mesh with cell diameter $2h$, because a uniform mesh with cells twice as large very naturally nests into the original uniform mesh, illustrated in figure 5-5 for a 2D mesh. It should be emphasised that it is not necessary for the meshes in a multigrid hierarchy to be nested, but the nesting property does simplify the construction of intergrid operators.

The smoothing step is used to reduce the high frequency components of the error on the given mesh. An inexpensive way to do this is to use a fixed number of iterations of an iterative solver. A good choice of smoother is problem dependent, but relaxation methods are usually chosen. We saw in section 5.1 that Jacobi and SOR methods are computationally cheap, but do not necessarily converge quickly. But the smoother does not need to iterate the relaxation method to convergence, just enough to reduce the error. The reason for this is that a method like Jacobi or SOR reduces the high frequency components of the error very efficiently, as we saw in example 5.4.1.

The calculation of the system matrix A_{2h} is related to the intergrid transfer operators and can be done using the Galerkin product. Alternatively A_{2h} can be obtained by rediscretising the problem, in equation (5.17) we rediscretised example 5.4.1 to obtain A_{2h}on the next coarser level. Either method can be used to perform multigrid, as demonstrated in algorithm 5.3.

Algorithm 5.3 describes a recursive implementation of a multigrid V–cycle. Using recursion we repeatedly coarsen the mesh and move from a fine grid to a coarse grid each time we recurse. Once the mesh is suitably coarse, the system matrix will be small enough that it can be inexpensively factorised using an LU–factorisation.

In algorithm 5.3 the restriction and prolongation are implemented as matrix calculations involving restriction matrix R and the prolongation matrix P. This is different to the restriction and prolongation methods used in example 5.4.1.

Algorithm 5.3 outlines what is referred to as the multigrid V–cycle. The diagram in figure 5-5 can be simplified to figure 5-6, which demonstrates the order in which the three grids in the hierarchy are visited as algorithm 5.3 progresses. Figure 5-4 also has the same "V" shape. At each dot in figure 5-6 a number of smoother steps are performed before moving in the direction of the arrow to the next grid, using restriction operators to move downward to coarser meshes and prolongation operators to move upward to finer meshes.

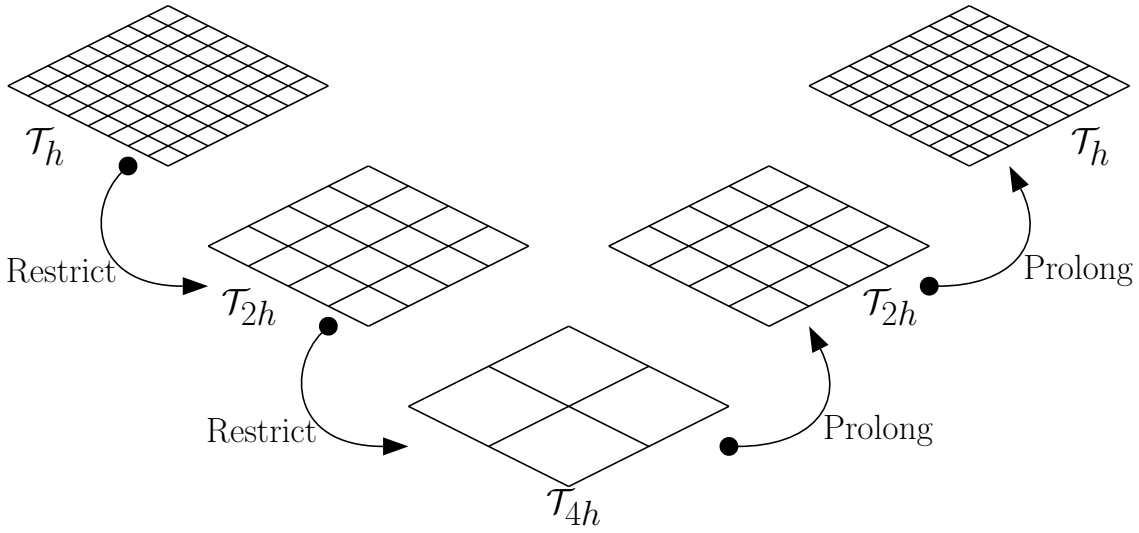

Figure 5-5: A 3 level multigrid hierarchy, where $\mathcal{T}_h$ represents a mesh with grid spacing h

Algorithm 5.3: Multigrid V–cycle smoothing α times before each restriction step and β times after each prolongation: Multigrid$(A_h, \underline{b}_h, h)$

Input : A_h – System matrix, $\underline{b}_h$ – Right-hand side,
$\underline{x}_0$ – Initial guess, h – Grid spacing
Output: $\underline{x}_h$
Initial guess $\underline{x}_0$
$\underline{x}_h = \text{Smooth}(A_h\underline{x}_h = \underline{b}_h$, initial guess $\underline{x}_0$, α times)
$\underline{r}_h = \underline{b}_h - A_h\underline{x}_h$
$\underline{r}_{2h} = R\underline{r}_h$ – Restriction step
Rediscretise A_{2h} OR Use Galerkin product $A_{2h} = RA_hP$
if *Mesh coarse enough* **then**
  $\underline{e}_{2h} = \text{Solve}(A_{2h}, \underline{r}_{2h})$ – with LU decomposition
else
  $\underline{x}_0 = \underline{0}$ – New initial guess
  $\underline{e}_{2h} = \text{Multigrid}(A_{2h}, \underline{r}_{2h}, \underline{x}_0, 2h)$ – Recursive step
end
$\underline{e}_h = P\underline{e}_{2h}$ – Prolongation step
$\underline{x}_h = \underline{x}_h + \underline{e}_h$
$\underline{x}_h = \text{Smooth}(A_h\underline{x} = \underline{b}$, initial guess $\underline{x}_h$, β times)
return $\underline{x}_h$

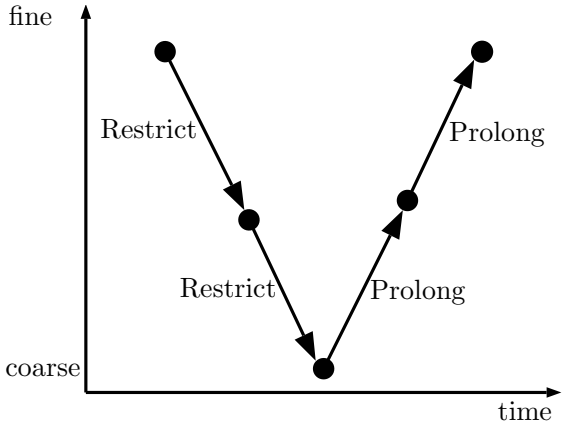

Figure 5-6: Graphical representation of a 3 level multigrid V–cycle

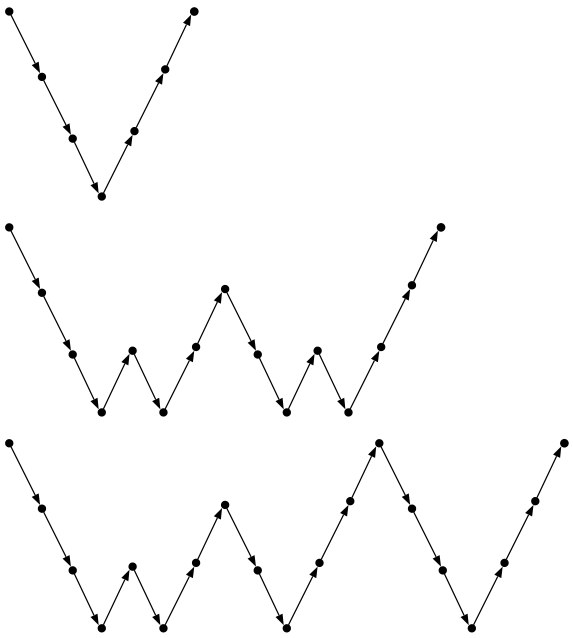

Figure 5-7: Multigrid cycles 4 level V–cycle, W–cycle, F–cycle

The V–cycle is not the only multigrid scheme. Figure 5-7 contains the V–cycle and two alternatives, the W–cycle and the F–cycle (or Full multigrid). More details of these alternative methods can be found in the books A multigrid tutorial [14] and Multigrid [54]. In the next chapter we will consider the multigrid algorithm in terms of bilinear forms.

CHAPTER 6

MULTIGRID

In chapter 5 we considered multigrid as a solver and preconditioner for a system of linear equations, but the theory of multigrid can be applied to the weak form also. In this chapter we see how multigrid theory can be applied to bilinear forms to obtain further insight into efficiently solving the shallow water equations.

Section 6.1 constructs the multigrid algorithm for a weak form. In section 6.2 we derive the weak form that must be solved when applying the non-nested multigrid method from Gopalakrishnan and Tan [31]. Finally, in section 6.3 we state explicitly the weak forms that we solve on the coarse space of our non-nested multigrid scheme and discuss the differences between the nested and non-nested algorithm.

6.1 Multigrid for a variational problem

Chapter 5 section 5.4.1 gives an overview of the GMG algorithm in terms of matrices, we now do the same for bilinear forms. Doing so we understand what the multigrid method in section 5.4 does in terms of function spaces and will later allow us to construct a non-nested multigrid algorithm. We consider an abstract problem in weak form, which is posed on the *fine* space. We seek $\phi_f \in \Phi_f$ such that

$$a_f(\phi_f, \psi_f) = b_f(\psi_f) \qquad \forall \psi_f \in \Phi_f, \tag{6.1}$$

where $a_f : \Phi_f \times \Phi_f \to \mathbb{R}$ is a bilinear form and $b_f : \Phi_f \to \mathbb{R}$ is a bounded linear functional. Φ is a general finite dimensional function space.

We make a clear distinction between functions which are in the primal space Φ_f and the space of all bounded linear functionals which form the dual space Φ_f^*. $\phi_f, \psi_f \in \Phi_f$ are primal functions and $b_f \in \Phi_f^*$ is a function in the dual space.

Using the notion of primal and dual functions we can discuss the construction of a prolongation operator. The weak form PDE problem on the *fine* space is equation (6.1) and the corresponding problem on the coarse space will be denoted with a subscript c.

Given an approximate solution ϕ_f, we define the residual as

$$r_f(\psi_f) := b_f(\psi_f) - a_f(\phi_f, \psi_f) \qquad \forall \psi_f \in \Phi_f. \tag{6.2}$$

Note that $r_f \in \Phi_f^*$, so to use the residual in equation (6.2) for the coarse space problem we will need to construct a restriction operator.

To construct the restriction operator, we start by defining the prolongation operator that takes coarse space primal functions to fine space primal functions.

$$\pi : \Phi_c \to \Phi_f. \tag{6.3}$$

We will drop brackets for the function π to emphasise that it is linear. Now if we have some function ϕ_c on the coarse space we can define a corresponding function on the fine space as $\phi_f := \pi\phi_c$. For nested function spaces, where $\Phi_c \subset \Phi_f$, the natural choice for the prolongation operator is simple injection, in which case $\phi_f = \phi_c$. This is not the only choice for the prolongation, as we see in section 6.3.

Constructing a restriction operator allows us to move from the fine space to the coarse space. We consider functions in the dual spaces to define

$$\rho^* : \Phi_f^* \to \Phi_c^*,$$

which is the transpose of the prolongation π. In other words, for a functional $r_c \in \Phi_f^*$, $\rho^*(r_c) = r_f \circ \pi \in \Phi_c^*$.

Now that we have the restriction operator we can transfer the fine space residual, defined in equation (6.2), to the coarse space

$$r_c = \rho^*(r_f).$$

This forms the right-hand side of the coarse problem.

The left-hand side of the coarse space problem is the bilinear form a_c, which is defined using the prolongation operator π in equation (6.3)

$$a_c(\phi_c, \psi_c) := a_f(\pi\phi_c, \pi\psi_c).$$

This is not the only way to define the problem on the coarse space, we could also have rediscretised the abstract problem on the coarse space. This is explored in section 6.3.

With a_c we can define the coarse space problem, which is: Find $\phi_c \in \Phi_c$ so that

$$a_c(\phi_c, \psi_c) = r_c(\psi_c) \qquad \forall \psi_c \in \Phi_c. \tag{6.4}$$

Since the right hand side of equation (6.4) is the (restricted) residual, the solution ϕ_c we obtain represents the error on the coarse space.

With the prolongation and restriction operators defined and a method for obtaining a coarse space problem (in weak form), everything required for the multigrid algorithm is specified. The two grid algorithm for bilinear forms is presented in algorithm 6.1.

The steps of the algorithm are almost identical to algorithm 5.3, but now formulated in terms of functions and bilinear forms. In algorithm 6.1 we have not specified how to perform the solve step, one choice is to use a multigrid solve extending the algorithm to work on as many levels as is desired.

6.2 Multigrid for the SWE

In this section we derive the weak problem that must be solved on the trace space when we use a HDG discretisation for the shallow water equations. The resulting system is of the form in equation (6.1) and is derived in theorem 6.2.1. We follow the argument laid out in section 2 of Cockburn et al. Multigrid for an HDG method applied to Poisson's equation [19].

Algorithm 6.1: 2–level multigrid for bilinear forms

Input : $a_f : \Phi_f \times \Phi_f \to \mathbb{R}$ fine space bilinear form,
$b_f : \Phi_f \to \mathbb{R}$ fine space linear functional
$\pi : \Phi_c \to \Phi_f$ prolongation (optional),
ϕ_0 initial guess

Output: $\phi_f \in \Phi_f$ approximate solution to $a_f(\phi_f, \psi_f) = b_f(\psi) \quad \forall \psi \in \Phi_f$

$\phi_f = \text{Smooth}(a_f(\phi_0, \psi) = b_f(\psi))$
$r_f = b_f(\psi) - a_f(\phi, \psi)$
if π *not specified* **then** π is simple injection;
Calculate the restriction ρ^* from π
$r_c = \rho^*(r_f)$ – restrict
Construct $a_c(\phi_c, \psi_c) := a_f(\pi\phi_c, \pi\psi_c)$ – coarse space bilinear form
$e_c = \text{Solve}(a_c(\phi_c, \psi_c) = r_c(\psi_c))$
$e_f = \pi(e_c)$ – prolong
$\phi_f = \phi_f + e_f$
$\phi_f = \text{Smooth}(a_f(\phi_f, \psi_f) = b_f(\psi_f))$
return ϕ_f

Our main result is contained in theorem 6.2.1, where we show that the bilinear form arising from the SWE is analogous to that for the Poisson equation. By "analogous to" we mean that the first term of our bilinear form appears to be the same as Cockburn et al., but requires slightly different lifting operators (section 6.2.3). Since we consider SWE and not Poisson our bilinear form also has an additional positive term.

We use different notation to Cockburn et al. as summarised in table 6.1.

Variable	Cockburn et al.	$\longrightarrow$	Our Notation
Vector trial function	$\vec{q}$	$\longrightarrow$	$\boldsymbol{U}$
Vector test function	$\vec{r}$	$\longrightarrow$	$\boldsymbol{V}$
Vector function space	V_h	$\longrightarrow$	$\mathcal{U}_h$
Scalar trial function	u	$\longrightarrow$	ϕ_S
Scalar test function	w	$\longrightarrow$	ψ
Scalar function space	W_h	$\longrightarrow$	Φ_h
Trace functions and function space	$\lambda, \mu, \eta \in M_h$	$\longrightarrow$	$\lambda, \mu \in \Lambda_h$
Operators	$\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{R}, \mathcal{S}, \mathcal{T}$ Unchanged		
Local operators *to* Vector function space	$\vec{\mathcal{Q}}_V, \vec{\mathcal{Q}}_W, \vec{\mathcal{Q}}$	$\longrightarrow$	$\mathcal{F}_\mathcal{U}, \mathcal{F}_\Phi, \mathcal{F}$
Local operators *to* Scalar function space	$\mathcal{U}_V, \mathcal{U}_W, \mathcal{U}$	$\longrightarrow$	$\mathcal{G}_\mathcal{U}, \mathcal{G}_\Phi, \mathcal{G}$

Table 6.1: Change of notation from paper by Cockburn et al. [19]

We also write the momentum equation of the SWE first, then the height potential, to be consistent with Cockburn et al. [19].

6.2.1 SWE problem

At each time step we solve the SWE problem that we specified in equations (2.4) and (2.5) on a periodic square domain

$$\rho_1 \boldsymbol{U} + \nabla(\phi_B \phi_S) = \boldsymbol{g} + \phi_S \nabla \phi_B \tag{6.5}$$
$$\rho_2 \phi_S + \nabla \cdot \boldsymbol{U} = f \tag{6.6}$$

for some known right hand side data $\boldsymbol{g}, f$ and where ρ_i are timestepping parameters. If we assume ϕ_B is constant for this chapter, we can simplify the first equation to

$$\rho_1 \boldsymbol{U} + \phi_B \nabla \phi_S = \boldsymbol{g}.$$

Writing $c = \phi_B^{-1}$ and re-scaling $\boldsymbol{g}$, we rewrite equation (6.5) as

$$c\rho_1 \boldsymbol{U} + \nabla \phi_S = \boldsymbol{g}. \tag{6.7}$$

Now let $\boldsymbol{V}$ and ψ be vector and scalar DG test functions respectively. We use the following inner product notation:

$$(\boldsymbol{U}, \boldsymbol{V}) := \sum_{K \in \mathcal{T}_h} \int_K \boldsymbol{U} \cdot \boldsymbol{V} \, dx$$
$$\langle \boldsymbol{U}, \boldsymbol{V} \rangle := \sum_{K \in \mathcal{T}_h} \int_{\partial K} \boldsymbol{U} \cdot \boldsymbol{V} \, dS \qquad \forall \boldsymbol{U}, \boldsymbol{V} \in \mathcal{U}_h$$

And the same notation for $\phi_S, \psi \in \Phi$:

$$(\phi_S, \psi) := \sum_{K \in \mathcal{T}_h} \int_K \phi_S \psi \, dx$$
$$\langle \phi_S, \psi \rangle := \sum_{K \in \mathcal{T}_h} \int_{\partial K} \phi_S \psi \, dS \qquad \forall \phi_S, \psi \in \Phi_h$$

This allows us to write the SWE, defined in equations (6.6) and (6.7), in weak form as: Seek $\boldsymbol{U} \in \mathcal{U}_h$ and $\phi_S \in \Phi_h$

$$(c\rho_1 \boldsymbol{U}, \boldsymbol{V}) + (\nabla \phi_S, \boldsymbol{V}) = (\boldsymbol{g}, \boldsymbol{V}) \qquad \forall \boldsymbol{V} \in \mathcal{U}_h$$
$$(\rho_2 \phi_S, \psi) + (\nabla \cdot \boldsymbol{U}, \psi) = (f, \psi) \qquad \forall \psi \in \Phi_h,$$

where the spaces Φ_h and $\mathcal{U}_h$ are defined in equation (3.28).

Integrating these equations by parts yields:

$$(c\rho_1 \boldsymbol{U}, \boldsymbol{V}) - (\phi_S, \nabla \cdot \boldsymbol{V}) + \langle \phi_S, \boldsymbol{V} \cdot \boldsymbol{n} \rangle = (\boldsymbol{g}, \boldsymbol{V}) \qquad \forall \boldsymbol{V} \in \mathcal{U}_h \tag{6.8}$$
$$(\rho_2 \phi_S, \psi) - (\boldsymbol{U}, \nabla \psi) + \langle \boldsymbol{U} \cdot \boldsymbol{n}, \psi \rangle = (f, \psi) \qquad \forall \psi \in \Phi_h \tag{6.9}$$

However, when we apply the DG method the ϕ_S and $\boldsymbol{U}$ in the angle brackets will not be well defined as they are double valued on the cell facet. For the hybridised scheme this problem is solved by introducing a trial function λ on the facets and corresponding test function $\mu \in \Lambda_h = \mathrm{Tr}(\Phi_h)$ (see section 4.1.1 for details of hybridisation). We define the upwind numerical flux to be

$$\widehat{\boldsymbol{U}} = \boldsymbol{U} + \tau_K (\phi_S - \lambda) \boldsymbol{n} \tag{6.10}$$

on the facets. Where τ_K is a flux parameter, which for the linear SWE $\tau_K = \sqrt{\phi_B}$.

We are restricting our discussion in section 6.2 to the upwind flux, we will discuss the Lax-Friedrichs flux in section 6.3.

Now the hybridised system corresponding to equations (6.8) and (6.9) is

$$\begin{aligned}(c\rho_1\boldsymbol{U},\boldsymbol{V}) - (\phi_S,\nabla\cdot\boldsymbol{V}) + \langle\lambda,\boldsymbol{V}\cdot\boldsymbol{n}\rangle &= (\boldsymbol{g},\boldsymbol{V}) \qquad \forall\boldsymbol{V}\in\mathcal{U}_h\\ (\rho_2\phi_S,\psi) - (\boldsymbol{U},\nabla\psi) + \left\langle\widehat{\boldsymbol{U}}\cdot\boldsymbol{n},\psi\right\rangle &= (f,\psi) \qquad \forall\psi\in\Phi_h\\ \left\langle\widehat{\boldsymbol{U}}\cdot\boldsymbol{n},\mu\right\rangle &= 0 \qquad \forall\mu\in\Lambda_h. \end{aligned} \tag{6.11}$$

6.2.2 Bilinear form of SWE

We can rewrite equation (6.11) by expanding the upwind flux in equation (6.10) and regrouping terms: Seek $(\boldsymbol{U},\phi_S,\lambda)\in\mathcal{U}_h\times\Phi_h\times\Lambda_h$ such that

$$\begin{array}{llll}(c\rho_1\boldsymbol{U},\boldsymbol{V}) & -(\phi_S,\nabla\cdot\boldsymbol{V}) & +\langle\lambda,\boldsymbol{V}\cdot\boldsymbol{n}\rangle & =(\boldsymbol{g},\boldsymbol{V}) \qquad \forall\boldsymbol{V}\in\mathcal{U}_h\\ (\boldsymbol{U},\nabla\psi) - \langle\boldsymbol{U}\cdot\boldsymbol{n},\psi\rangle & -(\rho_2\phi_S,\psi) - \langle\tau_K\phi_S,\psi\rangle & +\langle\tau_K\lambda,\psi\rangle & =(-f,\psi) \qquad \forall\psi\in\Phi_h\\ \langle\boldsymbol{U}\cdot\boldsymbol{n},\mu\rangle & +\langle\tau_K\phi_S,\mu\rangle & -\langle\tau_K\lambda,\mu\rangle & =0 \qquad \forall\mu\in\Lambda_h\end{array}$$

Note that we have multiplied the second equation by -1, in order to ensure the operator represented in equation (6.12) is self adjoint. We can write this in a more abstract form as

$$\begin{aligned}&\begin{cases}(\mathcal{A}\boldsymbol{U},\boldsymbol{V}) & +(\mathcal{B}^\top\phi_S,\boldsymbol{V}) & +\left\langle\mathcal{C}^\top\lambda,\boldsymbol{V}\cdot\boldsymbol{n}\right\rangle & =(\boldsymbol{g},\boldsymbol{V})\\ (\mathcal{B}\boldsymbol{U},\psi) & +(\mathcal{R}\phi_s,\psi) & +\left\langle\mathcal{S}^\top\lambda,\psi\right\rangle & =(-f,\psi)\\ \langle\mathcal{C}\boldsymbol{U},\mu\rangle & +\langle\mathcal{S}\phi_S,\mu\rangle & +\langle\mathcal{T}\lambda,\mu\rangle & =0\end{cases}\\ &\equiv:\begin{pmatrix}\mathcal{A} & \mathcal{B}^\top & \mathcal{C}^\top\\ \mathcal{B} & \mathcal{R} & \mathcal{S}^\top\\ \mathcal{C} & \mathcal{S} & \mathcal{T}\end{pmatrix}\begin{pmatrix}\boldsymbol{U}\\ \phi_S\\ \lambda\end{pmatrix} = \begin{pmatrix}\boldsymbol{g}_h\\ f_h\\ 0\end{pmatrix}.\end{aligned} \tag{6.12}$$

Where the operators of equation (6.12) are defined as follows:

$$\mathcal{A}\colon\mathcal{U}_h\to\mathcal{U}_h, \qquad (\mathcal{A}\boldsymbol{U},\boldsymbol{V}) := (c\rho_1\boldsymbol{U},\boldsymbol{V}).$$

Note that $\mathcal{A}^\top=\mathcal{A}$ (the operator $\mathcal{A}$ is self adjoint).

$$\begin{aligned}\mathcal{B}\colon\mathcal{U}_h\to\Phi_h, \qquad (\mathcal{B}\boldsymbol{U},\psi) &:= -(\nabla\cdot\boldsymbol{U},\psi)\\ &= (\boldsymbol{U},\nabla\psi) - \langle\boldsymbol{U}\cdot\boldsymbol{n},\psi\rangle\,.\end{aligned}$$

Hence, using the definition of adjoint:

$$\begin{aligned}\mathcal{B}^\top\colon\Phi_h\to\mathcal{U}_h, \qquad (\mathcal{B}^\top\phi_S,\boldsymbol{V}) &= (\phi_S,\mathcal{B}\boldsymbol{V})\\ &= -(\phi_S,\nabla\cdot\boldsymbol{V}).\end{aligned}$$

Also

$$\mathcal{C}\colon\mathcal{U}_h\to\Lambda_h, \qquad \langle\mathcal{C}\boldsymbol{U},\mu\rangle := \langle\boldsymbol{U}\cdot\boldsymbol{n},\mu\rangle\,,$$

so that

$$\begin{aligned}\mathcal{C}^\top\colon\Lambda_h\to\mathcal{U}_h, \qquad \left\langle\mathcal{C}^\top\lambda,\boldsymbol{V}\right\rangle &:= \langle\lambda,\mathcal{C}\boldsymbol{V}\rangle\\ &= \langle\lambda,\boldsymbol{V}\cdot\boldsymbol{n}\rangle\,.\end{aligned}$$

We also define

$$\mathcal{R}\colon \Phi_h \to \Phi_h, \qquad (\mathcal{R}\phi_S, \psi) := -(\rho_2 \phi_S, \psi) - \langle \tau_K \phi_S, \psi \rangle .$$

(Notice that $\mathcal{R}^\top = \mathcal{R}$). And

$$\mathcal{S}\colon \Phi_h \to \Lambda_h, \qquad \langle \mathcal{S}\phi_h, \mu \rangle := \langle \tau_K \phi_S, \mu \rangle .$$

Hence

$$\begin{aligned} \mathcal{S}^\top\colon \Lambda_h \to \Phi_h, \qquad \left\langle \mathcal{S}^\top \lambda, \psi \right\rangle &:= \langle \lambda, \tau_K \psi \rangle \\ &= \langle \tau_K \lambda, \psi \rangle . \end{aligned}$$

Finally,

$$\mathcal{T}\colon \Lambda_h \to \Lambda_h, \qquad \langle \mathcal{T}\lambda, \mu \rangle := -\langle \tau_K \lambda, \mu \rangle .$$

(Notice that $\mathcal{T}^\top = \mathcal{T}$). Given the above structure, and the observations above, the 3×3 matrix of operators is self adjoint.

The right-hand side functions, $\boldsymbol{g}_h$ and f_h in equation (6.12), are given by

$$\begin{aligned} \boldsymbol{g}_h &:= \boldsymbol{g} \\ f_h &:= -f. \end{aligned}$$

6.2.3 Local operators

We define local operators, which will act element-wise in the DG formulation and are essential for the reconstruction of the ϕ_S and $\boldsymbol{U}$ solutions form the trace variable λ. The local operators are

$$\begin{aligned} \mathcal{F}_\mathcal{U}\colon \mathcal{U}_h \to \mathcal{U}_h, \qquad \mathcal{F}_\Phi\colon \Phi_h \to \mathcal{U}_h, \\ \mathcal{G}_\mathcal{U}\colon \mathcal{U}_h \to \Phi_h, \qquad \mathcal{G}_\Phi\colon \Phi_h \to \Phi_h, \end{aligned}$$

such that

$$\begin{pmatrix} \mathcal{A} & \mathcal{B}^\top \\ \mathcal{B} & \mathcal{R} \end{pmatrix} \begin{pmatrix} \mathcal{F}_\mathcal{U} \boldsymbol{g}_h \\ \mathcal{G}_\mathcal{U} \boldsymbol{g}_h \end{pmatrix} = \begin{pmatrix} \boldsymbol{g}_h \\ 0 \end{pmatrix} \qquad \forall \boldsymbol{g}_h \in \mathcal{U}_h, \tag{6.13}$$

and

$$\begin{pmatrix} \mathcal{A} & \mathcal{B}^\top \\ \mathcal{B} & \mathcal{R} \end{pmatrix} \begin{pmatrix} \mathcal{F}_\Phi f_h \\ \mathcal{G}_\Phi f_h \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} \\ f_h \end{pmatrix} \qquad \forall f_h \in \Phi_h. \tag{6.14}$$

The operator $\left(\begin{smallmatrix} \mathcal{F}_\mathcal{U} & \mathcal{F}_\Phi \\ \mathcal{G}_\mathcal{U} & \mathcal{G}_\Phi \end{smallmatrix}\right)$ is constructed to form the Schur complement. If we construct the operator Schur complement in the bottom right of the matrix of operators in equation (6.12), we obtain the operator

$$\mathcal{T} - \begin{pmatrix} \mathcal{C} & \mathcal{S} \end{pmatrix} \begin{pmatrix} \mathcal{F}_\mathcal{U} & \mathcal{F}_\Phi \\ \mathcal{G}_\mathcal{U} & \mathcal{G}_\Phi \end{pmatrix} \begin{pmatrix} \mathcal{C}^\top \\ \mathcal{S}^\top \end{pmatrix}.$$

We see clearly here that $\left(\begin{smallmatrix} \mathcal{F}_\mathcal{U} & \mathcal{F}_\Phi \\ \mathcal{G}_\mathcal{U} & \mathcal{G}_\Phi \end{smallmatrix}\right)$ acts a like the inverse of $\left(\begin{smallmatrix} \mathcal{A} & \mathcal{B}^\top \\ \mathcal{B} & \mathcal{R} \end{smallmatrix}\right)$, which is the intuition behind the implicit definitions in equations (6.13) and (6.14).

Furthermore, since we are working with a hybridised formulation each of the block operators acts locally on each cell. This means that the application of $\left(\begin{smallmatrix} \mathcal{F}_\mathcal{U} & \mathcal{F}_\Phi \\ \mathcal{G}_\mathcal{U} & \mathcal{G}_\Phi \end{smallmatrix}\right)$ can be made very efficient, as it only works with local degrees of freedom (hence the name local

operators).

Further we define $\mathcal{F} : \Lambda_h \to \mathcal{U}$ and $\mathcal{G} : \Lambda_h \to \Phi_h$ such that for all $\mu \in \Lambda_h$

$$\begin{aligned}\mathcal{F}\mu &:= -\mathcal{F}_{\mathcal{U}}(\mathcal{C}^\top \mu) - \mathcal{F}_\Phi(\mathcal{S}^\top \mu)\\ \mathcal{G}\mu &:= -\mathcal{G}_{\mathcal{U}}(\mathcal{C}^\top \mu) - \mathcal{G}_\Phi(\mathcal{S}^\top \mu).\end{aligned} \tag{6.15}$$

Now we can apply $\left(\begin{smallmatrix}\mathcal{A} & \mathcal{B}^\top\\ \mathcal{B} & \mathcal{R}\end{smallmatrix}\right)$ to $\mathcal{F}\mu$ and $\mathcal{G}\mu$, firstly by expanding the operators as defined in the previous section, using equation (6.12):

$$\begin{aligned}\left(\begin{pmatrix}\mathcal{A} & \mathcal{B}^\top\\ \mathcal{B} & \mathcal{R}\end{pmatrix}\begin{pmatrix}\mathcal{F}\mu\\ \mathcal{G}\mu\end{pmatrix}, \begin{pmatrix}\boldsymbol{V}\\ \psi\end{pmatrix}\right) &= \begin{pmatrix}(\mathcal{A}\mathcal{F}\mu, \boldsymbol{V}) + (\mathcal{B}^\top \mathcal{G}\mu, \boldsymbol{V})\\ (\mathcal{B}\mathcal{F}\mu, \psi) + (\mathcal{R}\mathcal{G}\mu, \psi)\end{pmatrix}\\ &= \begin{pmatrix}(c\rho_1 \mathcal{F}\mu, \boldsymbol{V}) - (\mathcal{G}\mu, \nabla\cdot\boldsymbol{V}) & \\ -(\nabla\cdot\mathcal{F}\mu, \psi) - (\rho_2\mathcal{G}\mu, \psi) & -\langle \tau_K \mathcal{G}\mu, \psi\rangle\end{pmatrix}.\end{aligned} \tag{6.16}$$

Secondly, using the local operators as defined in equations (6.13) and (6.14):

$$\begin{aligned}\left(\begin{pmatrix}\mathcal{A} & \mathcal{B}^\top\\ \mathcal{B} & \mathcal{R}\end{pmatrix}\begin{pmatrix}\mathcal{F}\mu\\ \mathcal{G}\mu\end{pmatrix}, \begin{pmatrix}\boldsymbol{V}\\ \psi\end{pmatrix}\right) &= \left(\begin{pmatrix}\mathcal{A} & \mathcal{B}^\top\\ \mathcal{B} & \mathcal{R}\end{pmatrix}\left[\begin{pmatrix}-\mathcal{F}_{\mathcal{U}}\mathcal{C}^\top\mu\\ -\mathcal{G}_{\mathcal{U}}\mathcal{C}^\top\mu\end{pmatrix} + \begin{pmatrix}-\mathcal{F}_\Phi\mathcal{S}^\top\mu\\ -\mathcal{F}_\Phi\mathcal{S}^\top\mu\end{pmatrix}\right], \begin{pmatrix}\boldsymbol{V}\\ \psi\end{pmatrix}\right)\\ &= \left(\begin{pmatrix}-\mathcal{C}^\top\mu\\ 0\end{pmatrix} + \begin{pmatrix}0\\ -\mathcal{S}^\top\mu\end{pmatrix}, \begin{pmatrix}\boldsymbol{V}\\ \psi\end{pmatrix}\right)\\ &= \begin{pmatrix}-\langle \mu, \boldsymbol{V}\cdot\boldsymbol{n}\rangle\\ -\langle \mu, \tau_K\psi\rangle\end{pmatrix}\end{aligned} \tag{6.17}$$

Comparing terms in equations (6.16) and (6.17) we get the following relations:

$$(c\rho_1\mathcal{F}\mu, \boldsymbol{V}) - (\mathcal{G}\mu, \nabla\cdot\boldsymbol{V}) = -\langle \mu, \boldsymbol{V}\cdot\boldsymbol{n}\rangle \tag{6.18}$$

$$(\nabla\cdot\mathcal{F}\mu, \psi) + (\rho_2\mathcal{G}\mu, \psi) + \langle \tau_K(\mathcal{G}\mu - \mu), \psi\rangle = 0 \tag{6.19}$$

6.2.4 Reconstructing $\boldsymbol{U}$ and ϕ_S

Using the third line of equation (6.12)

$$\langle \boldsymbol{U}\cdot\boldsymbol{n}, \mu\rangle + \langle \tau_K\phi_S, \mu\rangle + \langle \tau_K\lambda, \mu\rangle = 0 \tag{6.20}$$

we obtain the trace problem. Once we have solved equation (6.20) we need to be able to reconstruct the momentum and height potential ($\boldsymbol{U}$ and ϕ_S), which give us the solution to the SWE, from the trace variable (λ). We do this by using the operators defined in equations (6.13) to (6.15).

If we start with the first two lines of the matrix of operators, we have the following equations: *If* we have already solved the λ, μ–system, and hence know λ, we can treat the λ terms in the above equations as data and write the system as

$$\begin{aligned}(\mathcal{A}\boldsymbol{U}, \boldsymbol{V}) \quad &+(\mathcal{B}^\top\phi_S, \boldsymbol{V}) \quad &=(\boldsymbol{g}_h, \boldsymbol{V}) - \left\langle \mathcal{C}^\top\lambda, \boldsymbol{V}\cdot\boldsymbol{n}\right\rangle\\ (\mathcal{B}\boldsymbol{U}, \psi) \quad &+(\mathcal{R}\phi_S, \psi) \quad &=(f_h, \psi) - \left\langle \mathcal{S}^\top\lambda, \psi\right\rangle.\end{aligned}$$

By construction of the local operators $\mathcal{F}$ and $\mathcal{G}$, and using linearity

$$\begin{aligned}\boldsymbol{U} &= \mathcal{F}_{\mathcal{U}}(\boldsymbol{g}_h - \mathcal{C}^\top\lambda) + \mathcal{F}_\Phi(f_h - \mathcal{S}^\top\lambda)\\ &= \mathcal{F}_{\mathcal{U}}\boldsymbol{g}_h + \mathcal{F}_\Phi f_h - \mathcal{F}_{\mathcal{U}}\mathcal{C}^\top\lambda - \mathcal{F}_\Phi\mathcal{S}^\top\lambda\\ &= \mathcal{F}_{\mathcal{U}}\boldsymbol{g}_h + \mathcal{F}_\Phi f_h + \mathcal{F}\lambda \end{aligned} \tag{6.21}$$

and

$$\begin{aligned}\phi_S &= \mathcal{G}_{\mathcal{U}}(\boldsymbol{g}_h - \mathcal{C}^\top\lambda) + \mathcal{G}_\Phi(f_h - \mathcal{S}^\top\lambda)\\ &= \mathcal{G}_{\mathcal{U}}\boldsymbol{g}_h + \mathcal{G}_\Phi f_h - \mathcal{G}_{\mathcal{U}}\mathcal{C}^\top\lambda - \mathcal{G}_\Phi\mathcal{S}^\top\lambda\\ &= \mathcal{G}_{\mathcal{U}}\boldsymbol{g}_h + \mathcal{G}_\Phi f_h + \mathcal{G}\lambda \end{aligned} \tag{6.22}$$

solves the above system. We have expressed in equations (6.21) and (6.22) the solution $\boldsymbol{U}$ and ϕ_S using only the right-hand side data $\boldsymbol{g}_h, f_h$ and the trace variable λ.

6.2.5 Trace problem

From equations (6.21) and (6.22) we know $\boldsymbol{U}$ and ϕ_S in terms of λ. Hence we can rewrite the trace problem, equation (6.20), as

$$\begin{aligned}&\langle(\mathcal{F}_{\mathcal{U}}\boldsymbol{g}_h + \mathcal{F}_\Phi f_h + \mathcal{F}\lambda)\cdot\boldsymbol{n}, \mu\rangle\\ &\quad + \langle\tau_K(\mathcal{G}_{\mathcal{U}}\boldsymbol{g}_h + \mathcal{G}_\Phi f_h + \mathcal{G}\lambda), \mu\rangle\\ &\quad - \langle\tau_K\lambda, \mu\rangle = 0\end{aligned}$$

Moving the data (f_h and $\boldsymbol{g}_h$) to the right hand side we obtain the trace system we must solve:

$$\begin{aligned}&\langle\mathcal{F}\lambda\cdot\boldsymbol{n}, \mu\rangle + \langle\tau_K\mathcal{G}\lambda, \mu\rangle - \langle\tau_K\lambda, \mu\rangle\\ &\qquad = -\langle\mathcal{F}_{\mathcal{U}}\boldsymbol{g}_h\cdot\boldsymbol{n}, \mu\rangle - \langle\mathcal{F}_\Phi f_h\cdot\boldsymbol{n}, \mu\rangle - \langle\tau_K\mathcal{G}_{\mathcal{U}}\boldsymbol{g}_h, \mu\rangle - \langle\tau_K\mathcal{G}_\phi f_h, \mu\rangle\,.\end{aligned} \tag{6.23}$$

This leads to the following result:

Theorem 6.2.1. *By simplifying equation* (6.23) *we obtain the weak problem: Find* $\lambda \in \Lambda_h$ *such that*

$$a(\lambda, \mu) = b(\mu) \quad \forall\mu \in \Lambda_h, \tag{6.24}$$

where

$$a(\lambda, \mu) := (c\rho_1\mathcal{F}\mu, \mathcal{F}\lambda) + (\rho_2\mathcal{G}\lambda, \mathcal{G}\mu) + \langle\tau_K(\mathcal{G}\lambda - \lambda), (\mathcal{G}\mu - \mu)\rangle$$

and

$$b(\mu) := -\langle\mathcal{F}\mu, \boldsymbol{g}_h\rangle - \langle\mathcal{G}\mu, f_h\rangle$$

Proof. Looking at the left hand side of equation (6.23) only, we can simplify the bilinear form that we must consider. The first term on the left is similar to the term in equation (6.18), but with $\mathcal{F}\lambda$ in place of $\boldsymbol{V}$:

$$\text{LHS} = -(c\rho_1\mathcal{F}\mu, \mathcal{F}\lambda) + (\mathcal{G}\mu, \nabla\cdot\mathcal{F}\lambda) + \langle\tau_K(\mathcal{G}\lambda - \lambda), \mu\rangle$$

Now the second term of this expression looks like equation (6.19), but μ replaced with λ and with ψ replaced with $\mathcal{G}\mu$:

$$\text{LHS} = (c\rho_1\mathcal{F}\mu, \mathcal{F}\lambda) - (\rho_2\mathcal{G}\lambda, \mathcal{G}\mu) - \langle\tau_K(\mathcal{G}\lambda - \lambda), \mathcal{G}\mu\rangle + \langle\tau_K(\mathcal{G}\lambda - \lambda), \mu\rangle$$

Changing sign everywhere, so that we have positivity, we can define our bilinear operator a to be

$$a(\lambda, \mu) := (c\rho_1 \mathcal{F}\mu, \mathcal{F}\lambda) + (\rho_2 \mathcal{G}\lambda, \mathcal{G}\mu) + \langle \tau_K(\mathcal{G}\lambda - \lambda), (\mathcal{G}\mu - \mu)\rangle \,.$$

Observe that $\left(\begin{smallmatrix} \mathcal{A} & \mathcal{B}^\top \\ \mathcal{B} & \mathcal{R} \end{smallmatrix}\right)$ is self-adjoint and therefore the local operators we defined in section 6.2.3 satisfy

$$\mathcal{F}_{\mathcal{U}}^\top = \mathcal{F}_{\mathcal{U}}, \qquad \mathcal{F}_{\Phi}^\top = \mathcal{G}_{\mathcal{U}}, \qquad \mathcal{G}_{\Phi}^\top = \mathcal{G}_{\Phi}.$$

Using this we can manipulate the right hand side of the above trace system we find that

$$\text{RHS} = \langle \mathcal{F}\mu, \boldsymbol{g}_h\rangle + \langle \mathcal{G}\mu, f_h\rangle$$

So (remembering to change the sign as we did for the bilinear form), we can define the corresponding linear functional as:

$$b(\mu) := -\langle \mathcal{F}\mu, \boldsymbol{g}_h\rangle - \langle \mathcal{G}\mu, f_h\rangle$$

QED

6.3 Non-nested multigrid

In the final section of this chapter we describe a method for performing multigrid on equation (6.24). The main difference between our method and the abstract method described in section 6.1 is that the function space on the coarse level will not be nested in the fine function space.

Algorithm 6.2 outlines a 2–level multigrid procedure for taking a general problem $a_1(\lambda, \mu) = b_1(\mu)$ on a fine space M_1 and transferring it to a coarse space M_0, where $M_0 \not\subset M_1$.

The inputs required for algorithm 6.2 are different to the inputs required for algorithm 6.1. In algorithm 6.1 we optionally could provide a prolongation operator. For algorithm 6.2 we *must* provide a prolongation $I_1 : M_0 \to M_1$.

We also require a suitable reformulation of the bilinear form a_1 on the coarse space. Whilst it is possible to use the method outlined in section 6.1 to automatically derive a coarse space bilinear form, doing so can be difficult. Computationally, it is more efficient to provide the bilinear form a_0 on the coarse space M_0 as input to the algorithm.

Although we have had to specify new inputs, the steps of the multigrid algorithm have not changed.

Now that we know the different inputs required to perform non-nested multigrid we must specify what these are. In section 4.1.1 we saw that is is possible to reduce the number of trace unknowns to one in the case of the upwind flux and two for the Lax-Friedrichs flux. For this reason we need to consider the non-nested multigrid schemes for each flux separately.

For the weak from given in equation (6.24) with *upwind* flux, the non nested spaces are the fine space $\Lambda_h = \text{Tr}(\Phi_h)$ and the coarse space that we will use is

$$M_0 = \{p \in H_1(\Omega) : \quad p|_K \in P^1(K) \quad \forall K \in \mathcal{T}_h\}$$

which is just the P^1 finite element space on Ω.

We know that $P^1 \not\subset \text{Tr}(\Phi_h)$ as functions in P^1 are defined on the interior of cells K and functions in $\text{Tr}(\Phi_h)$ are only defined on the facets of the mesh.

To move from coarse to fine space it suffices to just specify the prolongation, as we

Algorithm 6.2: 2–level non-nested multigrid

Input : $a_1 : M_1 \times M_1 \to \mathbb{R}$ fine space bilinear form,
$b_1 : M_1 \to \mathbb{R}$ fine space linear functional
$a_0 : M_0 \times M_0 \to \mathbb{R}$ coarse space bilinear form
$I_1 : M_0 \to M_1$ prolongation (not optional),
λ initial guess

Output: $\lambda \in M_1$ approximate solution to $a_1(\lambda, \mu) = b_1(\mu) \quad \forall \mu \in M_1$

$\lambda = \text{Smooth}(a(\lambda, \mu) = b_1(\mu))$
$r_1 = b_1(\mu) - a_1(\lambda, \mu)$
Calculate the restriction P_0 from I_1
$r_0 = P_0(r_1)$ – restrict
$e_0 = \text{Solve}(a_0(p, q) = r_0(q))$
$e_1 = I_1(e_0)$ – prolong
$\lambda = \lambda + e_1$
$\lambda = \text{Smooth}(a_1(\lambda, \mu) = b_1(\mu))$
return λ

saw in section 6.1. We define the prolongation $I_1^{\text{up}} : P^1 \to \text{Tr}(\Phi_h)$ to be the restriction of P^1 functions to the facets of the mesh. For a function $q \in P^1$

$$I_1^{\text{up}}(q) := \begin{cases} q|_{\partial K} & \text{for } d > 0 \\ \Pi q & \text{for } d = 0, \end{cases}$$

where an L_2–projection Π is used in the case of degree zero HDG, because we require the prolongated function to be constant on facets in this case. This is the prolongation used by Gopalakrishnan and Tan [31] for the mixed Poisson equation.

In the upwind flux case we define the coarse space bilinear form for $q, p \in P^1$ as

$$a_0(q, p) := \left(\frac{\phi_B}{\rho_1}\nabla q, \nabla p\right) + (\rho_2 q, p).$$

The parameter ϕ_B is the bathymetry term and ρ_1, ρ_2 parameters arise from the semi implicit timestepping of the SWE and equal some constant times Δt. The analysis required to find this form is similar to that used for the mixed Poisson formulation in Cockburn et al. [19]. Note that this bilinear form is the same as the bilinear that is obtained when solving the modified Helmholtz problem

$$-\nabla \cdot \nabla q + \frac{\rho_1 \rho_2}{\phi_B} q = k, \tag{6.25}$$

where k represents a calculated scalar valued right-hand side function.

An intuitive explanation for how the form is derived can be obtained by considering the continuous SWE as written in equations (6.6) and (6.7). In lemma 4.1.1 we saw that the trace variable $\widehat{\boldsymbol{U}}$ could be eliminated from the upwind flux. By taking the divergence of equation (6.7) and substituting into equation (6.6), we can eliminate the variable $\boldsymbol{U}$ from the SWE. Doing so we obtain

$$-\nabla \cdot \nabla \phi_S + \frac{\rho_1 \rho_2}{\phi_B}\phi_S = \frac{\rho_1}{\phi_B} f - \nabla \cdot \boldsymbol{g}.$$

Which has the same form as equation (6.25), but formulated in terms of the height poten-

tial variable rather than the trace variable. The mapping between the potential variable and the trace space ($\Phi_h \mapsto \mathrm{Tr}(\Phi_h)$) is handled by the implicitly defined local operators discussed in section 6.2.3. The full justification for the form of the trace problem follows Cockburn et al. [19].

Now we have specified everything we need to be able to use algorithm 6.2 when the SWE are numerically solved with the upwind flux.

In the case of the *Lax-Friedrichs* flux, the non-nested spaces are vector valued function spaces. These are the fine space $\Lambda_h = \mathrm{Tr}(\mathcal{U}_h)$ and the coarse space that we will use is the lowest order Raviart-Thomas function space

$$M_0 = \{\boldsymbol{q} \in H(\mathrm{div}, \Omega) : \quad \boldsymbol{q}|_K = \boldsymbol{\alpha}_K + \beta_K \boldsymbol{x} \quad \forall K \in \mathcal{T}_h\}, \tag{6.26}$$

where $H(\mathrm{div}, \Omega)$ denotes the space of functions with bounded weak divergence on the domain Ω and $\boldsymbol{\alpha}_K, \beta_K$ are constants on a cell K and are chosen such that $\boldsymbol{q}$ has a continuous normal derivative across all facets. We refer to equation (6.26) as the RT^1 space.

Just like $P^1 \not\subset \mathrm{Tr}(\Phi_h)$, $RT^1 \not\subset \mathrm{Tr}(\mathcal{U}_h)$ since functions in RT^1 are defined on the interior of cells K and functions in $\mathrm{Tr}(\mathcal{U}_h)$ are only defined on the facets of the mesh.

The prolongation $I_1^{\mathrm{LF}} : RT^1 \to \mathrm{Tr}(\mathcal{U}_h)$ is the L_2–projection of RT^1 functions on to the facets of the mesh. For a function $\boldsymbol{q} \in RT^1$

$$I_1^{\mathrm{LF}}(\boldsymbol{q}) := \boldsymbol{\lambda}$$

where $\boldsymbol{\lambda}$ satisfies

$$\int_{\partial K} \boldsymbol{\lambda} \cdot \boldsymbol{\mu} \; ds = \int_{\partial K} (\boldsymbol{q}|_K) \cdot \boldsymbol{\mu} \; ds \quad \forall \boldsymbol{\mu} \in \mathrm{Tr}(\mathcal{U}_h).$$

In the Lax-Friedrichs flux case we define the coarse space bilinear form for $\boldsymbol{q}, \boldsymbol{p} \in RT^1$ as

$$a_0(\boldsymbol{q}, \boldsymbol{p}) := -\left(\frac{\phi_B}{\rho_1} \nabla\nabla \cdot \boldsymbol{q}, \boldsymbol{p}\right) + (\rho_2 \boldsymbol{q}, \boldsymbol{p}).$$

This bilinear form is the same as the bilinear that is obtained when solving a "grad-div" problem of the form:

$$-\nabla\nabla \cdot \boldsymbol{q} + \frac{\rho_1 \rho_2}{\phi_B} \boldsymbol{q} = \boldsymbol{k} \tag{6.27}$$

Where $\boldsymbol{k}$ represents a calculated *vector* valued right-hand side function.

For an intuitive explanation of how to obtain this form, again consider the continuous SWE as written in equations (6.6) and (6.7). In lemma 4.1.2 we saw that the only the trace variable $\widehat{\phi}_S$ could be eliminated from the Lax-Friedrichs flux, leaving two degrees of freedom in the components of $\widehat{\boldsymbol{U}}$. By taking the gradient of equation (6.6) and substituting into equation (6.7) (the opposite to the upwind case), we can eliminate the variable ϕ_S from the SWE. Doing so we obtain

$$-\nabla\nabla \cdot \boldsymbol{U} + \frac{\rho_1 \rho_2}{\phi_B} \boldsymbol{U} = \rho_2 \boldsymbol{g} - \nabla f$$

Which has the same form as equation (6.27), but formulated in terms of the momentum rather than trace. Using a similar definition to that in section 6.2.3, we can construct a mapping between the momentum and the trace space ($\mathcal{U}_h \mapsto \mathrm{Tr}(\mathcal{U}_h)$).

Note that the "grad-div" differential operator has a non-trivial kernel, which makes this problem more difficult to solve numerically.

CHAPTER 7
IMPLEMENTATION

In this chapter we will focus on the tools used to numerically solve the shallow water equations. Chapter 3 presented the DG method, for obtaining a matrix equation from a PDE. In chapter 5 various methods of solving a matrix equation $Ax = b$ were discussed. Now we can put the two parts together and demonstrate how we use existing tools to implement solvers for the SWE and more importantly how to implement our custom preconditioning scheme combining hybridisation form chapter 4 and non-nested multigrid from chapter 6.

Whilst it is possible to implement a bespoke code from scratch to solve the SWE, this would take a great deal of time and effort. Additionally, writing bespoke code is prone to errors and requires time to debug before any numerical results can be obtained. Instead we want to test and demonstrate various preconditioning techniques with less effort. For our implementation and the results presented in chapter 8 we have chosen to use the Firedrake project [2, 44], an already existing software framework.

- We want to look at **efficient** solvers. Often solver libraries are the result of decades of continued development. It would be very difficult to make solvers as computationally efficient as the code produced by teams of researchers and experienced software engineers over a long period of time. By using the PETSc solvers [11] as part of Firedrake we have access to many high quality powerful solvers (including multigrid solvers).

- The solvers we look at should be suitable for **higher-order** DG discretisations. We saw in appendix A.2 that looking at problems in one spatial dimension and low order, basis construction and quadrature rules are quite simple. But, in two or three spatial dimensions and using high degree polynomial approximations, even just the basis construction gets more difficult. By using an already existing code base, we have access to arbitrary order polynomial approximations and a large suite of finite elements. Extensive testing ensures that the Firedrake project functions correctly. We also use the Slate language [29] that is part of Firedrake so we do not have to perform the hybridisation ourselves.

- Finally, we want this code to run efficiently on **modern architectures**, which means we need the code to perform well in parallel over multiple modern computer processors. The task of writing parallel code, whilst retaining efficiency and gaining performance over multiple Central Processing Units (CPUs) is not straightforward.

We use an existing project that is written from the ground up to run in parallel, so we only need to worry about the performance of the solver code that we implement ourselves. Firedrake is designed in such a way that it works seamlessly in parallel, the user does not have to take extra steps or make calls to parallelisation libraries themselves. Automatic code generation is used by Firedrake so that optimisation can be performed on the source code before it is compiled. This allows for very fine grained parallelisation[1] as well as coarse grained optimisation[2].

Firedrake allows the end user to express their problem in Python and automatically generates and runs high performance parallel C code to solve the problem numerically. By using Firedrake we save time implementing the finite element framework. Unit tests give assurances that Firedrake's code is correct, efficient and works in parallel. Implementing solvers for the SWE in Firedrake required the following steps:

The weak form of the SWE must be expressed in the framework. We will look at both the linear and non-linear SWE and we want to look at arbitrary degree polynomial approximations to the solution on various 2D meshes. This means we need to ensure the code we write is very general and works for all of these cases.

In order to test that we have implemented our weak form correctly and have obtained the correct discretisation, we can look at analytic solutions to the SWE. One such test is the "lake at rest" where the potential height of the water is a fixed constant on the domain and no velocity field is specified. The test can be applied with or without bathymetry, in both cases the water should not move and the lake should remain at rest. Another test case is the stationary vortex test case outlined in chapter 2, where the Coriolis force is balanced by the momentum of the spinning vortex of water. The vortex is a useful test case that allows us to obtain convergence results and analyse the error in the numerical solution obtained.

With correctly functioning code we can solve the SWE using existing solvers and preconditioning techniques. Firedrake exposes an interface to PETSc, which contains an extensive range of solvers, we can establish a performance baseline by trying various appropriate solvers already available in PETSc. We can then implement the approximate Schur complement factorisation from section 5.2.2 using PETSc options and the non-nested multigrid preconditioner, from section 6.3, combined with hybridisation, from section 4.1, using the Slate language to compare.

We implement the problem in a generic and object oriented way using Python, which allows running tests over a range of parameters and test cases. For this chapter we only outline a simple script in listings 7.2, 7.3, 7.5 and 7.7 that can solve the SWE. This allows us to demonstrate the power of the various features of Firedrake, without printing our entire code base.

[1] vectorisation and threading
[2] intra- and inter- node communication

7.1 Firedrake

Firedrake is a numerical PDE framework that uses the Python programming language to solve PDEs that are expressed in weak form. This is achieved by using automatic code generation to write, optimise, compile and run the kernels. Python is an interpreted object oriented programming language which we can use to express the PDE problem without having to compile the code before running it.

To see what is involved in this process we look at a Poisson problem on the two dimensional domain Ω:

$$\nabla^2 \phi = -f$$

Which in weak form can be written

$$\int_\Omega \nabla\phi \cdot \nabla\psi \; dx = \int_\Omega f\psi \; dx \tag{7.1}$$

If we seek a conforming finite element solution, we don't have to consider a numerical flux. We can implement the code that produces the kernel for the Poisson problem from equation (7.1):

```
1  from firedrake import *
2
3  # Mesh
4  mesh = UnitSquareMesh(10, 10)
5
6  # Function space and functions
7  V = FunctionSpace(mesh, "CG", 1)
8  phi = TrialFunction(V)
9  psi = TestFunction(V)
10
11 # Specify data
12 f = Function(V)
13 x, y = SpatialCoordinate(mesh)
14 f.interpolate((1+8*pi*pi)*sin(x*pi*2)*sin(y*pi*2))
15
16 # Bilinear form
17 a = ( dot(grad(phi), grad(psi)) ) * dx
18 # Linear operator
19 L = f * psi * dx
```

Listing 7.1: Firedrake code for Poisson problem

The part of the code that sets up the kernel is line 17, which looks very similar to the maths in equation (7.1).

A key philosophy of the Firedrake project is separation of concerns. This means that the end user should only need to be familiar with enough knowledge to be able to express a weak form of a PDE in the UFL language. The end user does not need to be an expert in mesh generation, or know how to optimise parallel code for a HPC architecture, these aspects have already been implemented in Firedrake by a domain specific expert. The separation of concerns is also inherent in the composability of Firedrake: The PETSc developer's primary concern is writing efficient solvers, whereas the Firedrake developer's primary concern is finite element assembly. This is a huge advantage, as the code that a Firedrake end user writes has the potential to be just as performant as a bespoke code from written from scratch by a team of experts.

The weak form of the PDE problem, in our case the shallow water equations, is first expressed in UFL (section 7.2) and then the two stage form compiler (TSFC) takes this,

generates and compiles efficient C code to assemble a finite element matrix using PETSc data structures. This, along with the user provided solver options, is passed to PETSc (section 7.4) to numerically solve the SWE. One advantage of Firedrake is that the problem is expressed at a high level of abstractions, this allows the form compiler to optimise by applying different transformations to the code [32]. This is a rather simplified overview of the code generation process, but the entire Firedrake toolchain, and most of its core components is illustrated in figure 7-1.

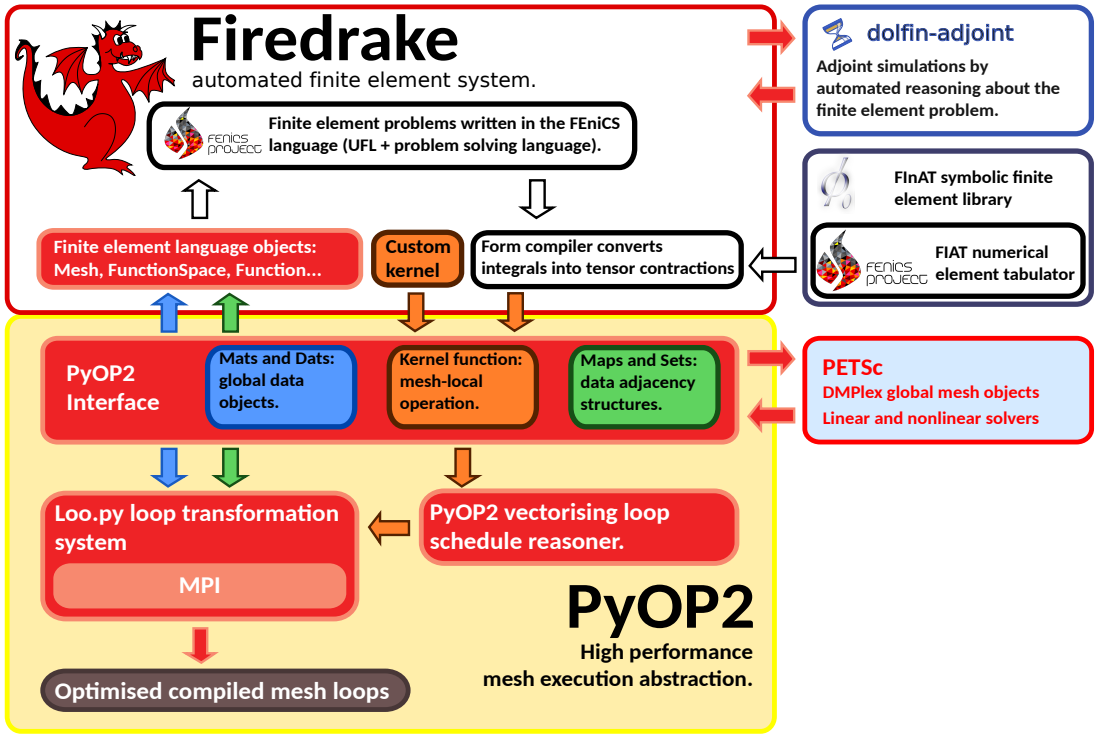

Figure 7-1: The Firedrake toolchain. Graphic courtesy of David Ham, currently not formally published

Two of the key components in figure 7-1 are PyOP2 and loo.py. PyOP2 [45] executes computational kernels over the mesh in parallel. The domain specific language is used internally by Firedrake to efficiently schedule the execution of the kernels for performant parallel code. This takes care of the coarse grained parallelism mentioned in the introduction. The fine grained parallelism is handled by loo.py [36], which is a relatively new addition to Firedrake [53]. The loo.py tool performs loop transformations on the automatically generated kernel code to ensure that once it is compiled it is highly efficient, taking full advantage of wide vector registers and SIMD parallelisation on CPUs.

There are numerous other features of Firedrake project that make it well suited for implementing an efficient solver for the SWE. These features are taken directly from the Firedrake website [2]:

> “
> - *Sophisticated automatic optimisation, including sum factorisation for high order elements, and vectorisation.*
> - *Geometric multigrid.*
> - *Customisable operator preconditioners.*

- *Support for static condensation, hybridisation, and HDG methods.*

”

These bullet points are precisely the ingredients of the efficient solver we want to construct.

Throughout the rest of the chapter, in four parts, we will illustrate the Firedrake code required to implement a simplified solver for the SWE. Each part is indicated in bold in the caption under the code listing. The weak form of the SWE is repeated in section 7.2.

```
1  from firedrake import *
2
3  # Number of elements and degree
4  elts = 40
5  degree = 3
6
7  # Timestepping parameters
8  T = 2.0
9  dt = 1/400
10 theta = 0.5
11
12 # Specify mesh
13 mesh = PeriodicSquareMesh(elts, elts, 4.0, quadrilateral=True)
14
15 # Define function spaces
16 Phi = FunctionSpace(mesh, 'DG', degree)
17 U = VectorFunctionSpace(mesh, 'DG', degree)
18 PhiU = Phi*U
19
20 # Initialise functions
21 phi_s = Function(Phi)
22 u = Function(U)
23
24 q = Function(PhiU)
25 p = TestFunction(PhiU)
26 q_next = TrialFunction(PhiU)
```

Listing 7.2: Firedrake, setting up constants, mesh, function spaces and functions. **Part 1 of the Python script**

The first part of this script is in listing 7.2. Here we hard code various constants that will be used for this script, in the true implementation, many of these are automatically calculated, but for this example they are hand calculated and hard coded. For instance we hard code the number of cells that are in the mesh, we want 40 cells in each spatial direction, and this is stored on the variable `elts`. We also hard code the degree polynomial degree for the DG space as `degree=3` and the total time for timestepping as `T=2`. Using these parameters we can hand calculate a stable timestep size and we set this in the variable `dt=1/400`.

In listing 7.2 we set up the mesh in line 13, which we will solve the SWE on as a periodic square mesh of quadrilaterals over the domain $\Omega = [0, 4] \times [0, 4]$.

We then set up DG function spaces that use polynomials of degree 3 in lines 16 to 17. We denoted in chapter 3 as Φ_h for the scalar valued function space and $\mathcal{U}_h$ for the vector valued function space. This is reflected in the code by naming the corresponding variables `Phi` and `U`. The product function space $\Phi_h \times \mathcal{U}_h$ can also be constructed in line 18. With these function spaces defined we can make general functions as well as special test and

trial functions in lines 21 to 26, that will allow for the expression of the SWE PDE, which is done using UFL.

7.2 UFL

The Unified Form Language (UFL) [7] is developed as part of the FEniCS project, but has since been used to express weak forms in other projects [1], including Firedrake. Its purpose in the Firedrake toolchain is for expressing weak forms in computer code, it is a Domain Specific Language (DSL) built on top of Python.

UFL itself does not try to convert the problem in weak form into low level code, it is an abstraction for expressing what is written mathematically. It is then the job of a *form compiler* to take the mathematical expression and generate the kernels for assembling the system matrix from a bilinear form or vectors from linear functionals. The Firedrake form compiler is TSFC [32]. What form compiling means in practise is that the finite element matrix and right-hand side vector are assembled from the bilinear form and linear functional, in the same manner as we did in chapter 3, and then passed to PETSc so that a solver can be used to compute the solution. In listing 7.3 lines 34 to 38 we use UFL to define the initial condition for the SWE problem and in lines lines 45 to 73 we define the bilinear form that we must use for timestepping.

The ultimate goal of the script outlined in this chapter is to step the SWE forward in time using a semi-implicit theta timestepper. The semi-implicit timesteppers are discussed in detail in section 4.2 and the implementation for the script is discussed in section 7.5. In order to perform this timestepping we need to write the bilinear form and linear functional as we did in equation (3.36), which is what we will express in UFL.

Before we start timestepping, we need to specify an initial condition. Here we choose a two dimensional Gaussian bump, centred on the coordinate $(2, 2)$ for the height potential, and $\mathbf{0}$ for the momentum. This is given by the functions:

$$\begin{aligned} \phi_s(\boldsymbol{x}, 0) &= 0.1 \times e^{-8\left[(x_1-2)^2+(x_2-2)^2\right]} \\ \boldsymbol{u}(\boldsymbol{x}, 0) &= \mathbf{0} \end{aligned} \tag{7.2}$$

Compare this to the code in lines 34 to 38 of listing 7.3, which is almost identical to the mathematical expression in equation (7.2). This demonstrates the power of UFL, one can almost immediately go from a mathematical formula to a representation of the expression in code.

UFL is also used to express the bilinear form that we derived in section 3.3. For this script we just consider the linear SWE, where the bathymetry is fixed and has constant value 1 across the domain. We use the Lax-Friedrich numerical flux. The linearised shallow water equations defined in equation (3.36), which we recall as:

$$\int_\Omega \frac{\partial \boldsymbol{q}_h}{\partial t} \cdot \boldsymbol{p} - \overline{F}(\boldsymbol{q}_h) : \nabla \boldsymbol{p} \, dx + \int_\Gamma [\overline{F^*}(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+)] : [\boldsymbol{p}^- \otimes \boldsymbol{n}^- + \boldsymbol{p}^+ \otimes \boldsymbol{n}^+] \, ds = \int_\Omega \boldsymbol{S}(\boldsymbol{q}_h) \cdot \boldsymbol{p} \, dx, \tag{7.3}$$

where $\boldsymbol{p}$ and $\boldsymbol{q}$ are test and trial functions respectively,

$$\overline{F} = \begin{pmatrix} U_1 & U_2 \\ \phi_S & 0 \\ 0 & \phi_S \end{pmatrix}, \qquad \boldsymbol{S}(\boldsymbol{q}) = \begin{pmatrix} 0 \\ fU_2 \\ -fU_1 \end{pmatrix} \tag{7.4}$$

```
30 # Coordinate system
31 x = SpatialCoordinate(mesh)
32
33 # Initial conditions
34 # Potential
35 ic = 0.1*exp(-8.0*( (x[0] - 2.0)**2 + (x[1] - 2.0)**2) )
36 phi_s.interpolate(ic)
37 # Momentum
38 u.assign(as_vector([0.0, 0.0]))
39
40 # Assign q to values of phi and u
41 q.sub(0).assign(phi_s)
42 q.sub(1).assign(u)
43
44 ## Bilinear form
45 def bilinear_form(q, p, sigma, dt):
46     phi_s, u = split(q)
47
48     # Mass terms
49     M = ( dot(q, p) )*dx
50
51     # Flux definition and differentiation term
52     flux = as_matrix([ [u[0],  u[1] ],\
53                        [phi_s, 0    ],\
54                        [0,     phi_s] ])
55     D = ( inner(flux, grad(p)) )*dx
56
57     # Numerical flux term
58     n = FacetNormal(mesh)
59     tau = Constant(1.0)
60     numerical_flux = avg(flux) + tau*avg(outer(q, n))
61     F = inner(numerical_flux, 2*avg(outer(p, n)) )*dS
62
63     # Source term
64     f_coriolis = Constant(1E-4)
65     source = as_vector([Function(Phi).assign(0.0),\
66                         +f_coriolis*u[1],         \
67                         -f_coriolis*u[0]          ])
68     S = ( dot(source, p) )*dx
69
70     # Final form(s)
71     L = - D + F - S
72
73     return M + Constant(sigma)*Constant(dt)*L
```

Listing 7.3: UFL defining initial conditions and bilinear form. **Part 2 of the Python script**

and

$$F^*\left(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+\right) = F^{\mathrm{LF}}\left(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+\right) = \{\{F(\boldsymbol{q}_h)\}\} + \frac{1}{2}|\tau|\left(\boldsymbol{q}_h^- \otimes \boldsymbol{n}^- + \boldsymbol{q}_h^+ \otimes \boldsymbol{n}^+\right) \tag{7.5}$$

is the Lax-Friedrich numerical flux and $\tau = 1$.

We can define the four bilinear forms

$$\begin{aligned}
\mathcal{M}(\boldsymbol{q}_h, \boldsymbol{p}) &:= \int_\Omega \boldsymbol{q}_h \cdot \boldsymbol{p}\ dx \\
\mathcal{D}(\boldsymbol{q}_h, \boldsymbol{p}) &:= \int_\Omega \overline{F}(\boldsymbol{q}_h) : \nabla \boldsymbol{p}\ dx \\
\mathcal{F}(\boldsymbol{q}_h, \boldsymbol{p}) &:= \int_\Gamma [\overline{F^*}(\boldsymbol{q}_h^-, \boldsymbol{q}_h^+)] : [\boldsymbol{p}^- \otimes \boldsymbol{n}^- + \boldsymbol{p}^+ \otimes \boldsymbol{n}^+]\ ds \\
\mathcal{S}(\boldsymbol{q}_h, \boldsymbol{p}) &:= \int_\Omega \boldsymbol{S}(\boldsymbol{q}_h) \cdot \boldsymbol{p}\ dx.
\end{aligned} \tag{7.6}$$

If we use the semi-implicit theta method, as we saw in section 4.2.1, for timestepping we can write equation (7.3) as

$$\Big[\mathcal{M} + \theta\Delta t \underbrace{(-\mathcal{D} + \mathcal{F} - \mathcal{S})}_{=:\mathcal{L}}\Big](\boldsymbol{q}_h^{(n+1)}, \boldsymbol{p}) = \Big[\mathcal{M} + (\theta - 1)\Delta t \underbrace{(-\mathcal{D} + \mathcal{F} - \mathcal{S})}_{=:\mathcal{L}}\Big](\boldsymbol{q}_h^{(n)}, \boldsymbol{p}). \tag{7.7}$$

The right-hand side of equation (7.7) uses the solution at the current timestep $\boldsymbol{q}_h^{(n)}$, which is known data, so the whole right-hand side is a linear functional. This means we can write equation (7.7) as

$$a(\boldsymbol{q}^{(n+1)}, \boldsymbol{p}) = b(\boldsymbol{p}), \tag{7.8}$$

where

$$a(\boldsymbol{q}, \boldsymbol{p}) = a(\boldsymbol{q}, \boldsymbol{p}\ ; \sigma, \Delta t) := \Big[\mathcal{M} + \sigma\Delta t\mathcal{L}\Big](\boldsymbol{q}, \boldsymbol{p}) \tag{7.9}$$

and

$$b(\boldsymbol{p}) = b(\boldsymbol{p}\ ; \boldsymbol{q}, \sigma, \Delta t) := \Big[\mathcal{M} + \sigma\Delta t\mathcal{L}\Big](\boldsymbol{q}, \boldsymbol{p}). \tag{7.10}$$

Since the left and right of equation (7.7) are so similar, we can use the same procedure to generate both the bilinear form in equation (7.9) and the linear form in equation (7.10). Algorithm 7.1 outlines the steps to do this.

Algorithm 7.1 defines all of the bilinear forms in equation (7.6) in turn. Each of these bilinear forms is written in terms of the flux function $\overline{F}$, the numerical flux $\overline{F}^*$ and the source function $\boldsymbol{S}$ which are defined in equations (7.4) and (7.5).

We can now use this algorithm to construct forms for equations (7.9) and (7.10). To generate a in equation (7.9) the four inputs of algorithm 7.1 are $\boldsymbol{q}^{(n+1)}, \boldsymbol{p}, \theta, \Delta t$, so the output will be bilinear form on the *left*-hand side of equation (7.7). If instead we input $\boldsymbol{q}^{(n)}, \boldsymbol{p}, (\theta - 1), \Delta t$ into algorithm 7.1, then we generate b in equation (7.10) and the output will be linear functional on the *right*-hand side of equation (7.7).

Algorithm 7.1 is implemented as a Python function to return each side in lines 45 to 73 of listing 7.3, where `M`, `D`, `F`, `S` and `L` correspond to the bilinear forms $\mathcal{M}, \mathcal{D}, \mathcal{F}, \mathcal{S}, \mathcal{L}$ in equation (7.6) and equation (7.7).

In the code `q_next` is a trial function and `p` is a test function so later, when the function is called in line 78 of listing 7.5, it creates the bilinear form `a` corresponding to a defined in equation (7.9). Whereas `q` is a general function so later, when the function is called in line 79 of listing 7.5, it creates the linear functional `b` corresponding to b defined in

Algorithm 7.1: Pseudo-code for generating bilinear form and linear functional in UFL

Input : $\boldsymbol{q}, \boldsymbol{p}$–functions, σ–weighting paramter, Δt–timestep size
Output: Generated form
ϕ_S, U_1, U_2 =ExtractFunctions($\boldsymbol{q}$)

$\mathcal{M}(\boldsymbol{q}, \boldsymbol{p}) := \int_\Omega \boldsymbol{q} \cdot \boldsymbol{p}\ dx$ – Define the mass term

$\overline{F}(\boldsymbol{q}) = \begin{pmatrix} U_1 & U_2 \\ \phi_S & 0 \\ 0 & \phi_S \end{pmatrix}$ – flux function
$\mathcal{D}(\boldsymbol{q}, \boldsymbol{p}) := \int_\Omega \overline{F}(\boldsymbol{q}) : \nabla \boldsymbol{p}\ dx$ – Define the differentiation term

$\boldsymbol{n}$ =UnitNormal(Cell)
$\tau = 1$
$\overline{F^*}(\boldsymbol{q}^-, \boldsymbol{q}^+) = \{\{F(\boldsymbol{q})\}\} + \frac{1}{2}|\tau|\,(\boldsymbol{q}^- \otimes \boldsymbol{n}^- + \boldsymbol{q}^+ \otimes \boldsymbol{n}^+)$ – numerical flux function
$\mathcal{F}(\boldsymbol{q}, \boldsymbol{p}) := \int_\Gamma [\overline{F^*}(\boldsymbol{q}^-, \boldsymbol{q}^+)] : [\boldsymbol{p}^- \otimes \boldsymbol{n}^- + \boldsymbol{p}^+ \otimes \boldsymbol{n}^+]\ ds$
– Define the numerical flux term

$f_{\text{Coriolis}} = 10^{-4}$
$\boldsymbol{S}(\boldsymbol{q}) = \begin{pmatrix} 0 \\ f_{\text{Coriolis}} U_2 \\ -f_{\text{Coriolis}} U_1 \end{pmatrix}$ – source function
$\mathcal{S}(\boldsymbol{q}, \boldsymbol{p}) := \int_\Omega \boldsymbol{S}(\boldsymbol{q}) \cdot \boldsymbol{p}\ dx$ – Define the source term

$\mathcal{L} = \mathcal{D} - \mathcal{F} + \mathcal{S}$ – Define the term $\mathcal{L}$
return $\mathcal{M} + \sigma \Delta t \mathcal{L}$

equation (7.10).

7.3 Slate

Firedrake provides functionality to manipulate the finite element matrices assembled by the form compiler at a high level using Slate [29]. The algebra performed by Slate uses the assembled matrices *local* to each cell, which avoids having to manipulate the global system matrix. Performing many small dense matrix operations allows us to generate more efficient code. Slate is a recent addition to the Firedrake project and is designed to be used for the hybridisation of PDE problems. It is therefore a very natural way of expressing the hybridised DG method that we described in section 4.1.

In section 4.1.2 we looked at the block structure of the global system matrix that is constructed when we perform a hybridised DG discretisation of the SWE. The matrix obtained has the following structure:

$$\Theta = \left(\begin{array}{c|c} \underline{\underline{\mathrm{M}}} + \Delta t \widehat{\underline{\underline{\mathrm{L}}}} & \Delta t \underline{\underline{\mathrm{G}}} \\ \hline \Delta t \underline{\underline{\mathrm{G}}}^{\top} & \Delta t \underline{\underline{\mathrm{T}}} \end{array} \right) \tag{7.11}$$

Where $\underline{\underline{\mathrm{M}}} + \Delta t \widehat{\underline{\underline{\mathrm{L}}}}$ has *no* coupling between cells and hence only contains cell local operations.

We also recall the Schur complement factorisation from section 5.2, which gives an *exact* formula for the inverse of a matrix in terms of its sub-blocks. The Schur complement factorisation solve is efficiently performed using algorithm 5.2.

To construct an algorithm to perform the Schur complement factorisation of a hybridised matrix we will consider a matrix Θ in equation (7.11) that represents a hybridised DG discretisation of a multi-field problem. We re-label the blocks to see the correspondence to the code in listing 7.4. We are now solving the matrix problem

$$\Theta \underline{x} = \underline{r}, \tag{7.12}$$

where $\underline{x}$ is the DOF solution vector and $\underline{r}$ is the right-hand side vector comging from the timestepping method.

We express the blocks of Θ in two different but equivalent ways

$$\Theta = \left(\begin{array}{c|c} \mathrm{M} & \mathrm{K} \\ \hline L & J \end{array}\right) = \left(\begin{array}{cc|c} A & B & C \\ D & E & F \\ \hline G & H & J \end{array}\right),$$

and do the same for $\underline{x}$ and $\underline{r}$

$$\underline{x} = \begin{pmatrix} \underline{\boldsymbol{q}} \\ \underline{\lambda} \end{pmatrix} = \begin{pmatrix} \underline{\phi} \\ \underline{\boldsymbol{U}} \\ \underline{\lambda} \end{pmatrix} \qquad \underline{r} = \begin{pmatrix} \underline{r}_{\boldsymbol{q}} \\ \underline{r}_{\lambda} \end{pmatrix} = \begin{pmatrix} \underline{r}_{\phi} \\ \underline{r}_{\boldsymbol{U}} \\ \underline{r}_{\lambda} \end{pmatrix}.$$

We start by considering equation (7.12) as:

$$\left(\begin{array}{c|c} M & K \\ \hline L & J \end{array}\right) \begin{pmatrix} \underline{\boldsymbol{q}} \\ \underline{\lambda} \end{pmatrix} = \begin{pmatrix} \underline{r}_{\boldsymbol{q}} \\ \underline{r}_{\lambda} \end{pmatrix}$$

From section 5.2 we can construct the Schur complement in the bottom right block $S_\lambda = J - LM^{-1}K$. Recall that M has a block diagonal inverse and so the Schur complement S_λ is a sparse matrix. We can use Slate operations to construct a symbolic expression for S_λ

```
S_lambda = J - L*M.inv*K,
```

which we use in line 21 of listing 7.4. Looking at algorithm 5.2 we see that we must perform three solves:

- Solve $M\tilde{\underline{r}} = \underline{r}_{\boldsymbol{q}}$
- Solve $S_\lambda \lambda = -L\tilde{\underline{r}} + \underline{r}_\lambda$
- Solve $M\underline{\boldsymbol{q}} = \underline{r}_{\boldsymbol{q}} - K\underline{\lambda}$

Slate allows the user to symbolically manipulate the matrices that consist of only cell local operations. This means we can write M^{-1} as `M.inv` in the code and this will construct cell local operations automatically. This allows us to express the three solves above as two solves:

1. Solve $S_\lambda \underline{\lambda} = -LM^{-1}\underline{r}_{\boldsymbol{q}} + \underline{r}_\lambda$
2. Solve $M\underline{\boldsymbol{q}} = \underline{r}_{\boldsymbol{q}} - K\underline{\lambda}$

We can go a step further and write the second solve as:

$$\begin{pmatrix} A & B \\ D & E \end{pmatrix} \begin{pmatrix} \underline{\phi} \\ \underline{\boldsymbol{U}} \end{pmatrix} = \begin{pmatrix} \underline{r}_\phi - C\underline{\lambda} \\ \underline{r}_{\boldsymbol{U}} - F\underline{\lambda} \end{pmatrix}. \tag{7.13}$$

We again construct the Schur complement in the bottom right to get $S_{\boldsymbol{U}} = E - DA^{-1}B$. Using Slate operations we construct a symbolic expression for $S_{\boldsymbol{U}}$,

```
S_u = E - D*A.inv*B,
```

which we use in line 40 of listing 7.4. The two solve steps from the Schur complement algorithm for equation (7.13) (algorithm 5.2) are:

a) Solve $S_{\boldsymbol{U}}\underline{\boldsymbol{U}} = -DA^{-1}\underline{r}_\phi + \underline{r}_{\boldsymbol{U}} - (F - DA^{-1}C)\underline{\lambda}$

b) Solve $A\underline{\phi} = \underline{r}_\phi - B\underline{\boldsymbol{U}} - C\underline{\lambda}$

We have written A^{-1} as it contains only cell local operations and we know this can be represented symbolically in Slate as `A.inv` and this performs only cell local operations.

So our algorithm should generate four things, a matrix and modified right-hand side for the problem reduced to the trace variables written in item 1, a symbolic expression to recover $\underline{\boldsymbol{U}}$ (item a) and a symbolic expression to recover $\underline{\phi}$ (item b). The pseudo-code to do this is given in algorithm 7.2, where we have used a function, 'ExtractBlocks()', which is aware of the block structure of Θ, to extract the correct number of blocks at each stage of the algorithm.

Algorithm 7.2: Pseudo-code for generating Slate expressions

Input : Θ – block matrix, $\underline{r}$ – block residual
Output: S_λ – matrix on trace block, $\hat{\underline{r}}$ – modified right-hand side
$\boldsymbol{U}_{\text{expr}}, \phi_{\text{expr}}$ – expressions for recovering solution

$\left(\begin{array}{c|c} M & K \\ \hline L & J \end{array}\right) = \text{ExtractBlocks}(\Theta, 2)$
$S_\lambda = J - LM^{-1}K$
$\begin{pmatrix} \underline{r}_{\boldsymbol{q}} \\ \underline{r}_\lambda \end{pmatrix} = \text{ExtractBlocks}(\underline{r}, 2)$
$\hat{\underline{r}} = \underline{r}_\lambda - LM^{-1}\underline{r}_{\boldsymbol{q}}$
$\left(\begin{array}{cc|c} A & B & C \\ D & E & F \\ \hline G & H & J \end{array}\right) = \text{ExtractBlocks}(\Theta, 3)$
$S_{\boldsymbol{U}} = E - DA^{-1}B$
$\begin{pmatrix} \underline{r}_\phi \\ \underline{r}_{\boldsymbol{U}} \\ \underline{r}_\lambda \end{pmatrix} = \text{ExtractBlocks}(\underline{r}, 3)$
$\underline{x} = \text{SymbolicVariable}$
$\begin{pmatrix} \underline{\phi} \\ \underline{\boldsymbol{U}} \\ \underline{\lambda} \end{pmatrix} = \text{ExtractBlocks}(\underline{x}, 3)$
$\boldsymbol{U}_{\text{expr}} =$"Solve$(S_{\boldsymbol{U}}\underline{\boldsymbol{U}} = -DA^{-1}\underline{r}_\phi + \underline{r}_{\boldsymbol{U}} - (F - DA^{-1}C)\underline{\lambda})$"
$\phi_{\text{expr}} =$"Solve$(A\underline{\phi} = \underline{r}_\phi - B\underline{\boldsymbol{U}} - C\underline{\lambda})$"
return $S_\lambda, \hat{\underline{r}}, \boldsymbol{U}_{\text{expr}}, \phi_{\text{expr}}$

The code in listing 7.4 illustrates how the Slate language can be used to break apart a problem with the same block structure as Θ. The code constructs a Schur complement on the trace space and defines symbolic expressions for reconstructing the solution, just as in algorithm 7.2. In listing 7.4 `a_form` is a bilinear form, just like the bilinear form returned by the Python function `bilinear_form` in listing 7.3.

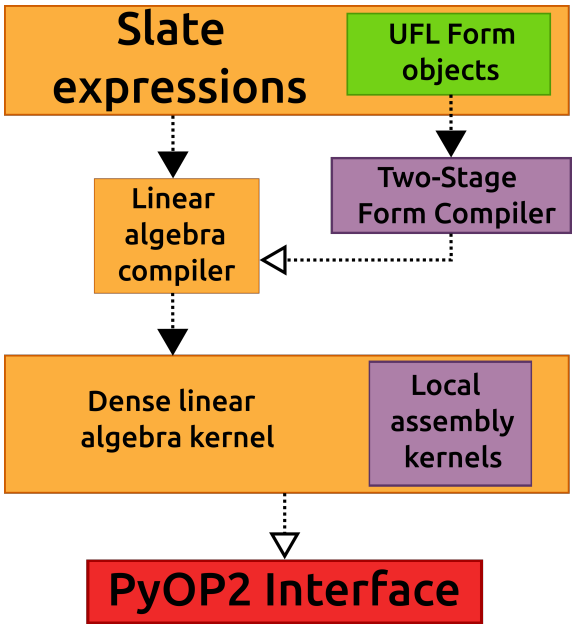

Figure 7-2: The Slate toolchain, from [29]

Figure 7-2 illustrates how Slate fits into the Firedrake toolchain. We take a UFL expression that defines a hybridised system of PDEs and manipulate the individual blocks of the resulting system matrix. Using Slate we exploit the block structure of the problem to form the Schur complement on the trace unknowns, which can be solved using PETSc. Slate also automatically determines what the cell local matrices are, constructs the small dense matrices and reconstructs the solution by performing multiple independent solves.

```
# This operator has the form:
# | A  B | C |   |       |   |
# | D  E | F | = |   M   | K |
# |------|---|   |-------|---|
# | G  H | J |   |   L   | J |
# NOTE: It is often the case that D = B.T,
# G = C.T, H = F.T, and J = 0, but we're not making
# that assumption here.
_O = Tensor(a_form)
Op = _O.blocks

# Extract sub-blocks:
# M = | A B |, K = | C |, L = | G H |, and J
#     | D E |      | F |
# which has block row indices (0, 1) and block
# column indices (0, 1) as well.
M = Op[:2, :2]; K = Op[:2, 2]
L = Op[2, :2] ; J = Op[2, 2]

# Schur complement for traces
S_lambda = J - L * M.inv * K

# Create mixed function for residual computation.
# This projects the non-trace residual bits into
# the trace space:
# -L * M.inv * r_q
_R = AssembledVector(residual)
R = _R.blocks
r_q = R[:2]
r_lambda = R[2]
r_lambda_hat = r_lambda - L * M.inv * r_q

# Reconstruction expressions
phi, u, lambda_ = solution.split()

# Local tensors needed for reconstruction
A = Op[0, 0]; B = Op[0, 1]; C = Op[0, 2]
D = Op[1, 0]; E = Op[1, 1]; F = Op[1, 2]

S_u = E - D * A.inv * B

r_phi, r_u, _ = residual.split()

# Solve locally using LU (with partial pivoting)
u_expr = S_u.solve(AssembledVector(r_u) -
                   D * A.inv * AssembledVector(r_phi) -
                   (F - D*A.inv*C) * AssembledVector(lambda_),
                   decomposition="PartialPivLU")

phi_expr = A.solve(AssembledVector(r_phi) -
                    B * AssembledVector(u) -
                    C * AssembledVector(lambda_),
                    decomposition="PartialPivLU")

return S_lambda, r_lambda_hat, u_expr, phi_expr
```

Listing 7.4: Slate

7.4 PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) [9, 11] is an extensive collection of linear and non-linear solvers for scientific applications. It specifically includes sparse solvers, ideal for solving the systems of equations that typically arise from the discretisation of PDEs. It also includes a unified set of data structures for vectors and matrices. PETSc is used by many finite element packages [10], including Firedrake, to solve the underlying linear (and non-linear) systems that arise from numerically solving PDEs.

PETSc itself is written in the C language, although it does have a Python interface: petsc4py [23]. Firedrake generates the PETSc data structures for solving the matrix problems. These are generated by the form compiler and are then passed to the PETSc solvers through the petsc4py interface. So by using the PETSc data structures, Firedrake has access to the whole PETSc library of solvers.

PETSc is designed and written to be used in parallel, which means that most solver routines and data structures work in parallel without modification.

```
77 # Set the left and right hand sides
78 a = bilinear_form(q_next, p, theta, dt)
79 b = bilinear_form(q, p, (theta - 1), dt)
80
81 # Setup some solvers to speed things up
82 params = {'ksp_type': 'cg', 'ksp_atol': 1E-8}
83 q_step = Function(PhiU)
84 problem = LinearVariationalProblem(a, b, q_step)
85 solver = LinearVariationalSolver(problem, solver_parameters=params)
```

Listing 7.5: Solver code. **Part 3 of the Python script**

Listing 7.5 demonstrates how Firedrake sets up solver options for PETSc and passes the linear problem to PETSc. We first set up the left and right hand sides using the function defined in listing 7.3. The solvers we set up in listing 7.5 will solve equation (7.8). Recall from listing 7.2, `q_next` is a `TrialFunction` and `p` is a `TestFunction`, so `a` is a bilinear form. But, `q` is a (known) `Function`, making `b` a linear functional, which can be passed to the function to obtain the right-hand side of equation (7.8).

We set up a `LinearVariationalProbem` in line 84 and solve into a new variable `q_step` at each time step. The solver is then constructed from the problem and the solver options. Once constructed the solver can be called as many times as we need, without having to reassemble the system matrix derived from the bilinear form. Multiple calls to the solver can be seen in section 7.5, listing 7.7.

Each of the solvers in PETSc's library of solvers can be controlled with a range of options. These options can either be passed at the command line, or as a dictionary of parameters through the Firedrake solver function call. In listing 7.5 they are stored in the variable `params` on line 82. We can also combine together different solvers, which is one idea we outlined briefly in section 5.2. One example of this is in the second set of GMG options in listing 7.6, where the `richardson` solver is used as a smoother and combined with the `mg` (PETSc's GMG solver). Better examples of PETSc solver composability is outlined in section 7.6.

The code in listing 7.6 shows how various solver options can be specified using a Python dictionary. The first two sets of parameters commented as `# Regular CG` and `# Preconditioned CG` show how to select the conjugate gradient solver. If no precon-

```
# Regular CG
solver_param = {'ksp_type': 'cg',
                'ksp_atol': 1E-5,
                'pc_type' : 'none'}
# Preconditioned CG
solver_param = {'ksp_type': 'cg',
                'ksp_atol': 1E-5}
# AMG
solver_param = {'ksp_type': 'preonly',
                'ksp_atol': 1E-5,
                'pc_type' : 'gamg'}
# GMG
solver_param = {'ksp_type': 'preonly',
                'ksp_atol': 1E-5,
                'pc_type' : 'mg',
                'mg_levels_ksp_type'  : 'richardson',
                'mg_levels_ksp_max_it': 3,
                'mg_levels_pc_type'   : 'sor'}
```

Listing 7.6: Various simple PETSc solver options

ditioner is specified, Firedrake will try and automatically precondition the problem. By default, if no parameters are provided, Firedrake will specify GMRES as the solver and precondition this using an incomplete LU–factorisation of the system matrix.

The parameters commented as `# AMG` shows how algebraic multigrid can be used as a solver. In PETSc `ksp_type` sets the *iterative* solver, which could be the Jacobi or CG method, and `pc_type` sets the preconditioner. Setting the `ksp_type` to `preonly` ensures that the problem is solved by a single application of the specified preconditioner, without any outer solver iterations. It also demonstrates how AMG can be used as a black box solver if it is specified as the preconditioner for Krylov subspace method.

In contrast the lines commented `# GMG` show the solver options for a geometric multigrid solver. For these options to work, a mesh hierarchy must already exist, which Firedrake allows the user to do with the `MeshHierarchy` class. Whilst Firedrake will set some defaults for smoothers, it is possible to get better convergence by specifying these as solver options. The key to (geometric) multigrid convergence is the choice of smoother. In this example, 3 iterations of a Richardson method are applied on each level and the Richardson method itself is preconditioned using SOR.

7.5 Timestepping

Firedrake does not provide general purpose timesteppers to solve time dependent problems and we must implement our own. For our numerical experiments we will use explicit methods as well as other IMEX schemes, see section 4.2.

For the results in chapter 8 we use our own Python module containing the various timesteppers, but for the example script we will just demonstrate how the theta method can be implemented, without the IMEX framework. The theta method for a general time-dependent problem

$$\frac{d\underline{q}}{dt} + f(t, \underline{q}) = 0, \qquad \underline{q}(0) = \underline{q}^{(0)} \tag{7.14}$$

that we want to solve over some time interval $[0, T]$, using a suitably chosen timestep Δt,

can be written as

$$\underline{q}^{(n+1)} + \theta\Delta t f((n+1)\Delta t, \underline{q}^{(n+1)}) = \underline{q}^{(n)} + (\theta - 1)\Delta t f(n\Delta t, \underline{q}^{(n)}).$$

This equation is then solved at every timestep to get the solution at the next timestep $\underline{q}^{(n+1)}$ using $\underline{q}^{(n)}$, until the final time T is reached. Algorithm 7.3 outlines this procedure by using a simple loop over the timesteps. Here the solve can be done using the solver we set up in listing 7.5.

Algorithm 7.3: Theta method

Input : $f(t, \underline{q})$, θ – Timestep parameter, $q^{(0)}$ – Initial condition, T – Total time, Δt – Timestep size

Output: $\underline{q}^{(n)}, n = 0, \ldots, N = \lceil \frac{T}{\Delta t} \rceil$

$t = 0$

$n = 0$

while $t < T$ **do**

  LHS $= \underline{q}^{(n+1)} + \theta\Delta t f(t + \Delta t, \underline{q}^{(n+1)})$

  RHS $= \underline{q}^{(n)} + (\theta - 1)\Delta t f(t, \underline{q}^{(n)})$

  $\underline{q}^{(n+1)} =$ Solve(LHS = RHS)

  $t = t + \Delta t$

  $n = n + 1$

end

return $\underline{q}^{(n)}, n = 0, \ldots, N$

```
89 # Timestep variables
90 t = 0
91 step = 0
92
93 # Timestepping loop
94 while t <= T:
95     solver.solve()
96     q.assign(q_step)
97     t += dt
98     step += 1
```

Listing 7.7: Timestepping code. **Part 4 of the Python script**

The code in listing 7.7 is the final part of the example script for solving the SWE, where we implement the timestepping routine. This very closely resembles the procedure outlined in algorithm 7.3, but now instead of timestepping the general problem in equation (7.14), we timestep the matrix ODE in equation (4.17) (that was setup by Firedrake in the previous parts of the script).

By setting up the solver as we did in listing 7.5 we do not need to redefine it each iteration of the while loop. The Firedrake `solver` object handles setting the "LHS" and "RHS"expressions that are used in algorithm 7.3 each time the `solver.solve()` method is called. In the script (listing 7.5) "LHS" corresponds to the variable `a` and "RHS" corresponds to the variable `b`. When we assign the value of `q_step` to the variable `q`, this automatically updates the Python object `b`, which is the right-hand side of equation (7.8). We do not have to keep redefining `b`.

For the numerical results in chapter 8, we want to be able to use any of the explicit or

IMEX timesteppers discussed in section 4.2. Rather than code each timestepping method separately, we take a more general and abstract approach. Looking at algorithm 4.1, we can execute a wide range of different IMEX schemes, just by providing the Butcher tableau as the input to the algorithm. We can achieve the same level of flexibility in our implementation by using object oriented programming.

To start, we define an abstract interface to a `System` class. This `System` class represents the discretisation of a PDE, which in our case is a DG or HDG discretisation of the shallow water equations. To ensure we conform to the specifications of the `System` class, the `DGSWE` and `HDGSWE` classes inherit from the `System` class. When using object inheritance from an abstract class we must ensure all of the necessary methods are implemented.

Reviewing section 4.2.2, there are relatively few methods that the `System` class needs in order to be stepped by an IMEX timestepping method. We need to be able to apply the action of the matrices $\underline{\underline{\mathrm{M}}}$ and $\underline{\mathrm{L}}$ as well apply the non-linear function $\underline{\underline{\mathrm{N}}}(\cdot)$, these will be three methods included in the `System` class. These matrices are defined in section 3.3.4 for the DG method. We also need to be able to solve using the system matrix $\underline{\underline{\mathrm{M}}} + \alpha\Delta t\underline{\mathrm{L}}$ (where α is a parameter defined by the timestepper) and the DG mass matrix $\underline{\underline{\mathrm{M}}}$, these are two more methods included in the `System` class. Note that these five methods are sufficient to perform IMEX timestepping. These methods are also more than sufficient to perform explicit timestepping, which never requires a system matrix solve.

With the abstract methods of the `System` class specified we can write an abstract `TimeStepperIMEX` class. We implement algorithm 4.1 using only the methods of the `System` class, and not any other specialised `DGSWE` or `HDGSWE` methods. By doing so the timestepper is capable of performing timestepping on *any* object whose class inherits from the `System` class and provides a concrete implementation of all the abstract methods. In our code this means that we can use the same timestepper class to timestep both the DG and HDG discretisation of the SWE.

The specific timestepping objects used to implement each of the named timestepping methods[3] is then just a specialisation of the abstract `TimeStepperIMEX` class. This specialisation is summarised by the (dual) Butcher tableau for the method. So a timestepper class such as `IMEXTheta` only needs to inherit from the abstract `TimeStepperIMEX` class and specify the Butcher tableau, no further code is required.

Using this object oriented approach and inheriting from abstract classes means if we want to use a new timestepper that we have not discussed here, provided it can be written as a dual Butcher tableau, the tableau is all we need to implement the method.

In order to view all of the timesteps from a simulation, Firedrake supports writing VTK files that can be visualised using the Paraview tool [6]. A view of the output of the four part script in this chapter (listings 7.2, 7.3, 7.5 and 7.7)at time $t \approx 1$ can be seen in figure 7-3.

The four part script solves the linear shallow water equations in equation (7.3), with initial condition in equation (7.2). By outputting the solution at every timestep it is possible to check by eye that we are solving the SWE correctly and obtaining a sensible solution. In figure 7-3 we can see the ripples radiating from the centre of the domain, $(2, 2)$, which is where the initial Gaussian was centred. A better way to check that the solution is correct is to compare to an analytic solution and calculate the error. The error is calculated in section 8.5, where the rate of convergence is also shown.

[3] these are: Explicit Euler, Heun, SSPRK3, IMEXTheta, IMEXARS2(2,3,2), IMEXSSP2(3,2,2),

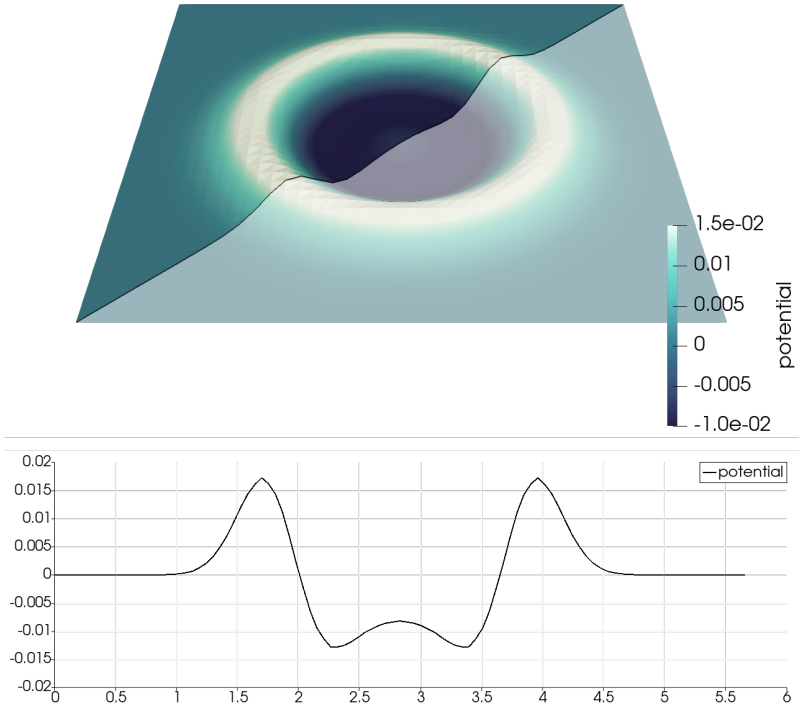

Figure 7-3: Potential height of simulation from script at $t \approx 1$ and height profile along the highlighted diagonal

7.6 Preconditioning

Putting together the code in listings 7.2, 7.3, 7.5 and 7.7 we have code that can solve the SWE. The script suffices for small simulations, but no effort has been made to precondition the problem or make the solver efficient.

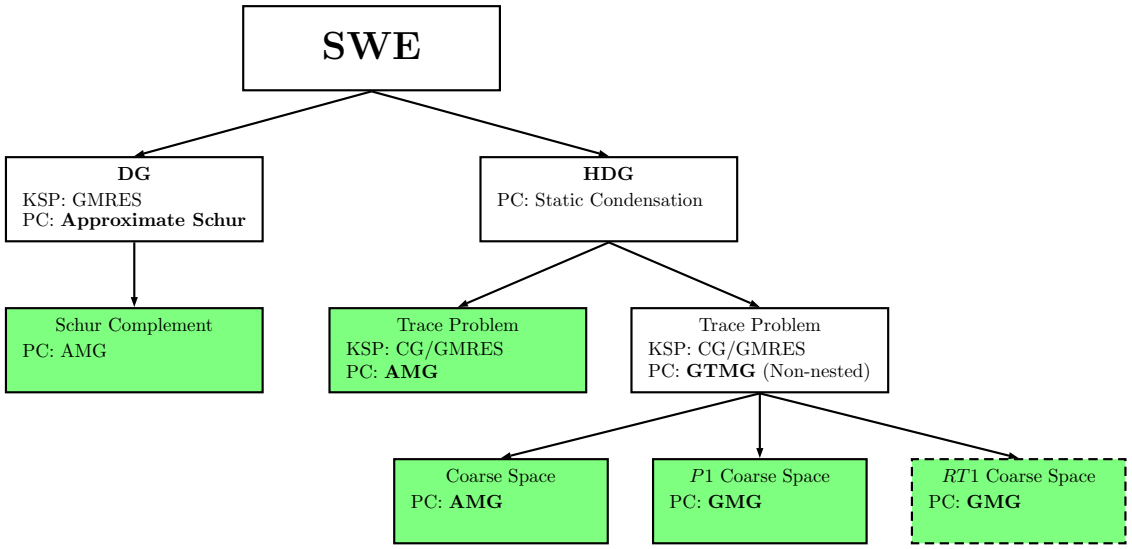

Figure 7-4: Tree of solver options

To construct an efficient solver we will compose together different solvers and preconditioners using PETSc. The composition of solvers is necessary as algorithms, such as the Schur complement factorisation algorithm (algorithm 5.2) and non-nested multigrid algorithm 6.2, contain a solve step. These solve steps also need preconditioning in order for them to converge quicky. It is only by combining together several different factorisations, solvers and preconditioners, can we tackle large systems with 10^9 degrees of freedom that we look at in chapter 8.

The tree in figure 7-4 breaks down the different strategies we use, each leaf of the tree is a separate solver strategy. On the first level of the tree we choose whether to perform a DG or HDG discretisation of the SWE, these have two different weak forms and are written differently in UFL.

If we go down the DG branch we use the weak form in equation (3.36), which yields the system matrix in equation (5.8) (and repeated in equation (7.15)). We can use the approximate Schur complement preconditioner described in section 5.2.2 for the outer GMRES solver. The approximate Schur complement preconditioner requires a solve on the Schur complement block, here we use AMG as the as described in section 5.4. The implementation of this solver is outlined in section 7.6.1.

If we instead go down the HDG branch we use the weak form in equation (4.2), which yields the system matrix in equation (5.13) (and repeated in equation (7.16)). When we perform a Schur complement factorisation on the hybridised system, as outlined in section 7.3, the Schur complement is formed in the block corresponding to the trace variable. We have two options for solving the trace problem.

One option is to use AMG as the trace solver, this strategy for solving the system is outlined in section 7.6.2.

The second option is to map the problem from the trace space to a P^1 space using the non-nested multigrid algorithm shown in algorithm 6.2. If we do that we then have a

choice of whether to use algebraic or geometric multigrid on the P^1 space. Both of these options are discussed in section 7.6.3.

The dashed box on the far right of figure 7-4 indicates another option. When the Lax-Friedrich flux is used with the HDG method, using non-nested multigrid to move to a P^1 space no longer works as the trace problem is vector valued. We can instead move to a Raviert-Thomas space as was described in section 6.3. This solver strategy is not fully explored here, but is briefly outlined in appendix A.4.

7.6.1 Approximate Schur complement

The first preconditioning technique we want to use is the approximate Schur complement that we analysed in section 5.2.2. We proved that this preconditioner should work well for lowest degree DG finite elements in section 5.2.2. Recall form the same section, that we looked at the matrix $\underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\mathrm{L}}}$:

$$A = \underline{\underline{\mathrm{M}}} + \Delta t \underline{\underline{\mathrm{L}}} = \left(\begin{array}{c|c|c} M + \Delta t Q_{\phi\phi} & \Delta t(-D_x + Q_{\phi U_1}) & \Delta t(-D_y + Q_{\phi U_2}) \\ \hline \Delta t(-D_{B,x} + Q_{U_1\phi} - M_{B,x}) & M + \Delta t Q_{U_1U_1} & \Delta t(Q_{U_1U_2} - fI) \\ \hline \Delta t(-D_{B,y} + Q_{U_2\phi} - M_{B,y}) & \Delta t(Q_{U_2U_1} + fI) & M + \Delta t Q_{U_2U_2} \end{array}\right) \tag{7.15}$$

The terms in this matrix are all defined in section 3.3.3. The solution strategy is outlined in figure 7-4, but to make this more concrete algorithm 7.4 gives the steps of the solver. We refer to the blocks of matrix A in equation (7.15) as

$$A = \left(\begin{array}{c|c} A_{\phi\phi} & A_{\phi \boldsymbol{U}} \\ \hline A_{\boldsymbol{U}\phi} & A_{\boldsymbol{UU}} \end{array}\right),$$

where the blocks

$$A_{\phi\phi} = \begin{pmatrix} M + \Delta t Q_{\phi\phi} \end{pmatrix} \qquad A_{\phi \boldsymbol{U}} = \begin{pmatrix} \Delta t(-D_x + Q_{\phi U_1}) & \Delta t(-D_y + Q_{\phi U_2}) \end{pmatrix}$$

$$A_{\boldsymbol{U}\phi} = \begin{pmatrix} \Delta t(-D_{B,x} + Q_{U_1\phi}) \\ \hline \Delta t(-D_{B,y} + Q_{U_2\phi}) \end{pmatrix} \qquad A_{\boldsymbol{UU}} = \left(\begin{array}{c|c} M + \Delta t Q_{U_1U_1} & \Delta t(Q_{U_1U_2} - fI) \\ \hline \Delta t(Q_{U_2U_1} + fI) & M + \Delta t Q_{U_2U_2} \end{array}\right).$$

When we form the Schur complement factorisation (equation (5.5)), the Schur complement is formed in the "$\phi\phi$ block" and is

$$S_{\phi\phi} = A_{\phi\phi} - A_{\phi \boldsymbol{U}} A_{\boldsymbol{UU}}^{-1} A_{\boldsymbol{U}\phi}.$$

The outer solve is done by a GMRES solver, since we are not guaranteed a symmetric positive definite system matrix using a DG discretisation. We know from chapter 5 that we can reduce the number of iterations by using a preconditioner, that must be applied in every iteration. The preconditioner used is the approximate Schur complement, the details of which are outlined in algorithm 5.2. That algorithm involves three solves with two distinct matrices which are the blocks of the system matrix. The Schur complement is formed in the top right as this is the smaller block matrix.

We then use two different solvers on each of these blocks. To obtain an approximation for $A_{\boldsymbol{UU}}$ block, we use a combination of block Jacobi and an incomplete LU factorisation. We expect the $S_{\phi\phi}$ solve to be more expensive as the Schur complement matrix is less sparse than $A_{\boldsymbol{UU}}$, as outlined in section 5.2. We use the powerful AMG solver on the Schur complement.

Algorithm 7.4: Approximate Schur complement factorisation preconditioning for solving a DG discretisation of the SWE

Input : $A \in \mathbb{R}^{n \times n}$, $\underline{b} \in \mathbb{R}^n$
Output: $\underline{y} \in \mathbb{R}^n$
$\underline{y}$ = **Solve** $A\underline{x} = \underline{b}$ using GMRES – outer solve
  for *Each iteration of GMRES* **do**
    if *residual* $\leq 10^{-8}$ **then** break;
    Precondition A with approx Schur complement factorisation on $\phi\phi$ block
      Solve $A_{\boldsymbol{UU}}$ equation with block Jacobi (ILU on those blocks)
      Solve $S_{\phi\phi}$ equation using AMG – inner solve
    end
  end
end
return $\underline{y}$

To recreate this algorithm in Firedrake we need to pass the correct parameters to PETSc, these are outlined in listing 7.8

```
# Approximate Schur complement preconditioner
solver_param = {'ksp_type': 'gmres',
                'ksp_rtol': 1E-8,
                'pc_type': 'fieldsplit',
                'pc_fieldsplit_type': 'schur',
                'pc_fieldsplit_schur_fact_type': 'FULL'
                'pc_fieldsplit_schur_precondition': 'selfp',
                'fieldsplit_0': {'ksp_type': 'preonly',
                                 'pc_type': 'bjacobi',
                                 'sub_pc_type': 'ilu'}},
                'fieldsplit_1': {'ksp_type': 'preonly',
                                 'pc_type': 'gamg',
                                 'mg_levels': {'ksp_type': 'chebyshev',
                                               'ksp_max_it': 2,
                                               'pc_type': 'bjacobi',
                                               'sub_pc_type': 'sor'}}
```

Listing 7.8: PETSc solver options for approximate Schur complement preconditioner

The outer solver we use is GMRES and is specified by the `ksp_type`. We will halt the GMRES algorithm when the reletive residual error is less than 10^{-8}.

The GMRES solver is preconditioned using a Schur factorisation, which in PETSc is a type of field splitting preconditioner. So termed because in the SWE we break apart the system matrix into the potential height and momentum fields. We explicitly tell PETSc to construct the *full* Schur complement factorisation, PETSc will allow us to use the upper, lower or diagonal blocks of the factorisation in equation (5.6) if we want a different preconditioner.

The approximation of $A_{\boldsymbol{UU}}$ is specified using `fieldsplit_1` parameters, we use a combination of block Jacobi (`bjacobi`) and an incomplete LU factorisation, (`ilu`).

PETSc never constructs the inverse of the Schur complement for the height potential block, instead the action of the inverse is applied using a solver. In this case we use AMG (`gamg`) as the inner solver, specified using `fieldsplit_0` paramaters. This combination of solver parameters produces a preconditioning technique equivalent to what is written

in algorithm 7.4 by using solver parameters available in PETSc.

Note 7.6.1. The diagonal approximation to the matrix $A_{\boldsymbol{UU}}$ represents what is possible in PETSc using *only* PETSc options, and represents a baseline to compare our hybridised method to. It may be possible to get a more competitive approximate Schur complement preconditioner by using a block diagonal approximation to the matrix $A_{\boldsymbol{UU}}$. There is currently no PETSc option that would give a preconditioner using a block diagonal approximation directly, so to test this we would need a custom preconditioner. In this thesis we investigate a custom preconditioner only for the hybridised formulation. A better approximate Schur complement preconditioner is a topic for future study.

7.6.2 Hybridisation

The second preconditioning technique, which we want to compare to the approximate Schur complement preconditioner, is using a hybridised DG method. This preconditioning technique is discussed in section 5.3 and aims to be a better preconditioner at higher order. We recall the matrix from section 5.3:

$$\begin{aligned}\widehat{A} &= \left(\begin{array}{c|c} \underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}} & \Delta t\underline{\underline{\mathrm{G}}} \\ \hline \Delta t\underline{\underline{\mathrm{G}}}^{\top} & \Delta t\underline{\underline{\mathrm{T}}} \end{array}\right) \\ &= \left(\begin{array}{c|c|c|c} M + \Delta t\widehat{Q}_{\phi\phi} & \Delta t(-D_x + \widehat{Q}_{\phi U_1}) & \Delta t(-D_y + \widehat{Q}_{\phi U_2}) & \Delta t G_\phi \\ \hline \Delta t(-D_{B,x} + \widehat{Q}_{U_1\phi} - M_{B,x}) & M + \Delta t\widehat{Q}_{U_1U_1} & \Delta t(\widehat{Q}_{U_1U_2} - fI) & \Delta t G_{U_1} \\ \hline \Delta t(-D_{B,y} + \widehat{Q}_{U_2\phi} - M_{B,y}) & \Delta t(\widehat{Q}_{U_2U_1} + fI) & M + \Delta t\widehat{Q}_{U_2U_2} & \Delta t G_{U_2} \\ \hline \Delta t G_\phi^{\top} & \Delta t G_{U_1}^{\top} & \Delta t G_{U_2}^{\top} & \Delta t\underline{\underline{\mathrm{T}}} \end{array}\right)\end{aligned} \tag{7.16}$$

The terms are defined in section 4.1.2. To make the block structure clear we have split the matrix $\underline{\underline{\mathrm{G}}}$ into three parts so that $\Delta t\underline{\underline{\mathrm{G}}} = (\Delta t G_\phi | \Delta t G_{U_1} | \Delta t G_{U_2})^{\top}$. For the solver in algorithm 7.5, we have used $\widehat{A}$ to make it clear that this matrix is the global system matrix for the HDG method and has different block structure to A in equation (7.15).

When we form the Schur complement factorisation (equation (5.5)), the Schur complement is formed in the "$\lambda\lambda$ block" and is

$$S_{\lambda\lambda} = \Delta t\underline{\underline{\mathrm{T}}} - (\Delta t)^2\underline{\underline{\mathrm{G}}}\,(\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}})^{-1}\underline{\underline{\mathrm{G}}}^{\top},$$

which is sparse as $\underline{\underline{\mathrm{M}}} + \Delta t\widehat{\underline{\underline{\mathrm{L}}}}$ is block diagonal.

In this strategy we first perform hybridisation, as we outlined in section 4.1. In the same way as we thought of the Schur complement factorisation in algorithm 7.4 as being a preconditioner, we can think of the reduction of the global system onto the trace system as a preconditioner here. Once we obtain the trace system requiring a $S_{\lambda\lambda}$ matrix solve, we can use GMRES as the solver and precondition using AMG. We could not use GMG as the solver or preconditioner here as the the problem depends on the degrees of freedom that are situated on the facets of the mesh and we do not have a mesh hierarchy for this skeleton mesh.

Once GMRES has converged and the trace system is solved, the final step of this algorithm is to reconstruct the global solution. This step is relatively inexpensive as it only uses small cell local matrices in the reconstruction.

To recreate this algorithm in Firedrake we use one of the inbuilt Firedrake preconditioners and the options specified in listing 7.9.

Algorithm 7.5: Hybridisation preconditioning using AMG on the trace block for solving a HDG discretisation of the SWE

Input : $\widehat{A} \in \mathbb{R}^{m \times m}$, $\underline{b} \in \mathbb{R}^m$
Output: $\underline{y} \in \mathbb{R}^m$
$\underline{z}$ =**Precondition** $\widehat{A}$ by reducing to problem on trace $\lambda\lambda$ block
  Solve $S_{\lambda\lambda}$ equation using GMRES – outer solve
    for *Each iteration of GMRES* **do**
      if *residual* $\leq 10^{-8}$ **then** break;
      Precondition $S_{\lambda\lambda}$ equation with AMG – inner solve
    end
  end
end
$\underline{y}$ = Reconstruct solution from $\underline{z}$
return $\underline{y}$

```
# Hybridisation preconditioner
solver_param = {'mat_type': 'matfree',
                'ksp_type': 'preonly',
                'pc_type': 'python',
                'pc_python_type': 'firedrake.SCPC',
                'pc_sc_eliminate_fields': '0, 1',
                'condensed_field': {'ksp_type': 'gmres',
                                    'pc_type': 'gamg',
                                    'mat_type': 'aij',
                                    'ksp_rtol': 1E-8,
                                    'mg_levels': {'ksp_type': 'chebyshev',
                                                  'ksp_max_it': 2,
                                                  'pc_type': 'sor'}}
```

Listing 7.9: PETSc solver options for hybridisation using SCPC

Reduction of the global problem to the trace system is done using the `firedrake.SCPC` preconditioner. This is where the Slate language is used to manipulate the blocks of the system matrix. The preconditioner performs the steps of algorithm 7.2 and the code is very similar to that seen in listing 7.4, but more generic so that it can be applied to more general problems.

The solver options for the trace space are specified in the `condensed_field` part of the solver options. Here we use GMRES as a solver by specifying the `ksp_type` and select AMG as a preconditioner by specifying `pc_type`.

The reconstruction of the global solution is handled automatically by `firedrake.SCPC`, so this step of the algorithm does not explicitly appear in the code that we write.

7.6.3 Non-nested multigrid

Finally, we want to combine the hybridisation with the Gopalakrishnan and Tan non-nested multigrid technique discussed in section 6.3, for a HDG discretisation using the *upwind* flux. The global matrix is the one that appears in equation (7.16). Note that using a P^1 coarse space, algorithm 7.6 and listing 7.10 only work when we use an upwind flux as our choice of numerical flux in the discretisation of the SWE. The Lax-Friedrich case is treated differently and mentioned briefly at the end of this section.

The non-nested multigrid solver extends the solver in algorithm 7.5, so the first part of algorithm 6.2 is the same. However, to perform the non-nested multigrid step we must know the prolongation $P^1 \to \mathrm{Tr}(\Phi_h)$. This is given by

$$I_1(p) := \begin{cases} p|_K & \text{for } d > 0 \\ \Pi p & \text{for } d = 0 \end{cases}.$$

We must also specify the bilinear form on the P^1 space, which is

$$a_0(p, q) := \left(\frac{\phi_B}{\rho_1}\nabla p, \nabla q\right) + (\rho_2 p, q), \tag{7.17}$$

for $p, q \in P^1$. Algorithm 6.2 provides the exact details of this multigrid method. The algorithm for the *solver* is outlined in algorithm 7.6.

The difference between algorithms 7.5 and 7.6 is the choice of preconditioner on the trace space. By using the non-nested multigrid the coarse space is P^1 and we can use either algebraic or geometric multigrid. This is indicated in the algorithm by "*multigrid".

The solver parameters for recreating this solver in firedrake ar given in listing 7.10. Reduction of the global problem uses the `firedrake.SCPC` preconditioner as in the previous section. Again we use GMRES as a solver on the `condensed_field`.

To precondition this GMRES solver we use our own custom preconditioner `firedrake.GTMGPC`[4]. We have to pass a few extra parameters using the application context (`appctx`) dictionary to to use the custom preconditioner. The `appctx` contains a `p1_callback` which is used to specify, using UFL, the weak form of the problem on the coarse space of the non-nested hierarchy. Also in the dictionary is a `get_p1_space` function which informs the preconditioner which function space to use on the coarse space and hence which space the functions used in `p1_callback` belong to. Optionally the `interpolation_matrix` can be passed through using the application context, although if not specified, Firedrake can determine this using its own interpolation code.

[4]Currently available in a branch of Firedrake, soon to be moved to master

Algorithm 7.6: Hybridisation preconditioning using non-nested multigrid preconditioning on the trace block for solving a HDG discretisation of the SWE

Input : $\widehat{A} \in \mathbb{R}^{m \times m}$, $\underline{b} \in \mathbb{R}^m$
Output: $\underline{y} \in \mathbb{R}^m$
$\underline{z}$ =**Precondition** $\widehat{A}$ by reducing to problem on trace $\lambda\lambda$ block
  Solve $S_{\lambda\lambda}$ equation using GMRES – outer solve
    for *Each iteration of GMRES* **do**
      if *residual* $\leq 10^{-8}$ **then** break;
      Precondition $S_{\lambda\lambda}$ with non-nested multigrid: Trace$\rightarrow P^1$
        Solve rediscretised problem on the P^1 coarse space using *multigrid
      end $P^1 \rightarrow$Trace
    end
  end
end
$\underline{y}$ = Reconstruct solution from $\underline{z}$
return $\underline{y}$

Once we have moved to P^1 space we can solve the P^1 problem as we like, this is indicated in listing 7.10 by the unspecified variable `mg_param`. Listings 7.11 and 7.12 both specify parameters that could be used for the coarse space solve.

AMG can again be used, the options are shown in listing 7.11.

Since non-nested multigrid moves the problem to a P^1 space we can also use GMG to solve equation (7.17), whereas this was not an option before on the trace space. If we are using a hybridised upwind numerical flux, we showed in section 6.3 that the problem on the P^1 space is given by equation (7.17). Suitable GMG solver options for `mg_params` are given in listing 7.12.

Notice that in both sets of multigrid solver options, listings 7.11 and 7.12, the smoother used is a combination of Chebyshev iterations and block Jacobi preconditioned SOR. This is to get around PETSc's lack of a truly parallel SOR. Chebyshev iterations are used as the solver, as this avoids having to compute inner products (which are expensive in parallel). Block Jacobi divides the sparse matrix into the same number of blocks as there are processors, and each processor performs SOR on its own block.

We also use the same smoother in place of a coarse grid solver. Normally at the coarsest level a direct method, such as an *LU* factorisation is used since the problem is small enough. Experimentally, it was determined that just smoothing on the coarsest level was enough to reduce the residual error by the desired amount. This also yields a faster method, as less parallel communication is required.

If we use the hybridised Lax-Friedrichs numerical flux, the calculations in section 6.2 no longer hold. We showed in section 4.1.1 that for the hybridised Lax-Friedrichs flux, the number of trace variables could be reduced to two by eliminating the $\widehat{\phi}_S$ trace variable.

In section 6.3 we show that the coarse space bilinear form is

$$a_0(\boldsymbol{q}, \boldsymbol{p}) := -\left(\frac{\phi_B}{\rho_1}\nabla\nabla\cdot\boldsymbol{q}, \boldsymbol{p}\right) + (\rho_2\boldsymbol{q}, \boldsymbol{p}). \tag{7.18}$$

for $\boldsymbol{q}, \boldsymbol{p} \in RT^1$. The operator for the coarse space problem contains the gradient of divergence, a differential operator that has a null space. Solving equation (7.18) with GMG is much more difficult, although it is positive definite and is better conditioned

```
# Hybridisation with GTMG
solver_params = {'mat_type': 'matfree',
                 'ksp_type': 'preonly',
                 'pc_type': 'python',
                 'pc_python_type': 'firedrake.SCPC',
                 'pc_sc_eliminate_fields': '0, 1',
                 'condensed_field':
                    {'ksp_type': 'gmres',
                     'mat_type': 'aij',
                     'ksp_monitor':None,
                     'pc_type': 'python',
                     'ksp_rtol': 1E-8,
                     'pc_python_type': 'firedrake.GTMGPC',
                     'gt': {'mat_type': 'aij',
                            'mg_levels':{'ksp_type': 'chebyshev',
                                         'ksp_max_it': 2,
                                         'pc_type': 'bjacobi',
                                         'sub_pc_type': 'sor'},
                            'mg_coarse': mg_param}}}

appctx = {'get_coarse_operator' : p1_callback,
          'get_coarse_space'    : get_p1_space,
          'interpolation_matrix': interpolation_matrix}
```

Listing 7.10: PETSc solver options for hybridisation with SCPC and GTMGPC

```
# AMG paramters
mg_param = {'ksp_type': 'preonly',
            'pc_type': 'gamg',
            'ksp_rtol': 1E-8,
            'pc_mg_cycles': 'v',
            'mg_levels': {'ksp_type': 'chebyshev',
                          'ksp_max_it': 2,
                          'pc_type': 'bjacobi',
                          'sub_pc_type': 'sor'},
            'mg_coarse': {'ksp_type':'chebyshev',
                          'ksp_max_it':2,
                          'pc_type':'sor'}}
```

Listing 7.11: PETSc solver options for AMG on coarse space of GTMGPC

in the large ρ_1, ρ_2 limit. A brief discussion of the solver options used to stabilise the "grad-div" operator in equation (7.18) can be found in appendix A.4.

```
# GMG paramters for Upwind flux
mg_param = {'ksp_type': 'preonly',
            'pc_type': 'mg',
            'ksp_rtol': 1E-8,
            'pc_mg_levels':2,
            'pc_mg_cycles': 'v',
            'mg_levels': {'ksp_type': 'chebyshev',
                          'ksp_max_it': 2,
                          'pc_type': 'bjacobi',
                          'sub_pc_type': 'sor'},
            'mg_coarse': {'ksp_type':'chebyshev',
                          'ksp_max_it':2,
                          'pc_type':'sor'}}
```

Listing 7.12: PETSc solver options for GMG on coarse space of GTMGPC

CHAPTER 8

RESULTS

In chapter 2 in equations (2.11) to (2.13) we gave an example a non-trivial analytic solution of the linear and non-linear SWE, which consists of a rotating vortex centred in our domain. Since we have this analytic expression for the solution at all timesteps, we use it here to verify that all the results in this section are producing the correct solution. In section 8.5 we measure rate of convergence of the methods, that is how quickly the difference between the true analytic solution and the computed solution decreases to zero as we increase the resolution. The initial conditions for all problems is the stationary vortex.

We consider two variants of the SWE summarised in table 8.1. The linear shallow water equations with constant bathymetry given in equation (3.36), in this case we use the upwind flux. This variant is our model problem, which is computationally easier to solve and lets us focus on the performance of the preconditioners discussed in section 7.6. The other variant is closer to a real world problem, we consider the fully non-linear SWE problem with bathymetry, given in equation (3.30). The non-linear equations necessitate the use of the Lax-Friedrichs flux. In both cases we use GMRES as the outer solver.

	Constant bathymetry	Non-constant bathymetry
Linear	Upwind flux and CG solver	-
Non-linear	-	Lax-Friedrich flux and GMRES solver

Table 8.1: SWE that we consider in the results chapter

For a given variant of SWE problem we will solve for a range of different mesh sizes and different polynomial degrees p for the DG space. For a unit square mesh with periodic boundary conditions, we choose a refinement r, then divide the square into 2^r sections in each dimension, resulting in $2^r \times 2^r$ square cells. Each square cell is divided along the diagonal into 2 triangles resulting in the $2 \times 2^r \times 2^r$ cells in the resulting mesh. The number of cells for each refinement is summarised in the first two columns of table 8.2. Each cell then contains $\frac{1}{2}(p+1)(p+2)$ degrees of freedom, and each of the 3 faces of each cell have $(p+1)$ DOFs, where p is the polynomial degree. We then have the scalar valued potential height ϕ and the two components of the momentum $\boldsymbol{U}$ we have triple the number of degrees of freedom. The total cell and facet DOFs for the selected degrees $p = 1, 3, 5$ are summarised in the final six columns of table 8.2. The the number of facet DOFs listed corresponds to the number of facet DOFs when using the upwind flux, twice as many are required for Lax-Friedrichs flux. The largest problems we solve (for the single

node tests) consist of a DOF vector $\underline{q}$ with the number of cell DOFs in the table. In total the largest problem we consider, the degree 3, refinement 8 system, has nearly 4 million DOFs before hybridisation.

degree	-	$p=1$		$p=3$		$p=5$	
refinement	# Cells	Cell	Facet	Cell	Facet	Cell	Facet
$r=4$	512	4 608	1 536	15 360	3 072	32 256	4 608
$r=5$	2 048	18 432	6 144	61 440	12 288	129 024	18 432
$r=6$	8 192	73 728	24 576	245 760	49 152	516 096	73 728
$r=7$	32 768	294 912	98 304	983 040	196 608	2 064 384	294 912
$r=8$	131 072	1 179 648	393 216	3 932 160	786 432	-	-

Table 8.2: Number of DOFs on cell interiors and facets for different mesh refinements and polynomial degrees

The implementation uses non-dimensionalised units for all quantities. All of the simulations are run for 0.5 non-dimensionalised time units, which corresponds to 12hrs in real time units.

The timestep size is calculated using the a heuristic, based on the CFL condition. First we calculate the maximum wave speed of the system given by

$$c_g = \frac{T_{\text{ref}}}{R_\oplus}\sqrt{gH},$$

where $T_{\text{ref}} = 8.64 \times 10^4$s the length of one day, $R_\oplus = 6.37 \times 10^6$m is the radius of the Earth, $g = 9.81\text{ms}^{-2}$ is the acceleration due to gravity and $H = 2 \times 10^3$m is the average depth of the ocean. The timestep is then chosen to be

$$\Delta t = \frac{\alpha \rho c_g \Delta x}{2p+1},$$

where α is a parameter set to 1 for an explicit timestepping method and 10 for an implicit timestepping method. Δx is the cell diameter and p is the polynomial degree. The parameter $\rho = 0.2$ is motivated by the work by Cockburn et al. [21] and is chosen such that the explicit Euler method is borderline stable.

The SWE is then iterated forward in time on the University of Bath HPC facility, *Balena*. Except for scaling runs or otherwise noted results are obtained with a single compute node (16×IvyBridge Xeon E5-2650v2 cores, 128Gb DDR3-1866 RAM). Each SWE problem had to fit into memory and complete reasonable time, so only the refinements and degrees in table 8.2 were run [1].

The various solver parameters are summarised in figure 8-1 and were discussed in section 7.6.

For brevity we will refer to the different combinations of solver options by the parts of the parameters highlighted in bold in figure 8-1. These are the options that make these solvers distinct from the others considered.

For all these results the solver tolerance was kept at 10^{-8} meaning that the residual is redduced by 8 orders of magnitude. The iteration counts presented are the number of

[1] Refinement 8, degree 5 ($\approx$ 6 million cell DOFs total) takes over an hour to run by itself, and is excluded for this reason

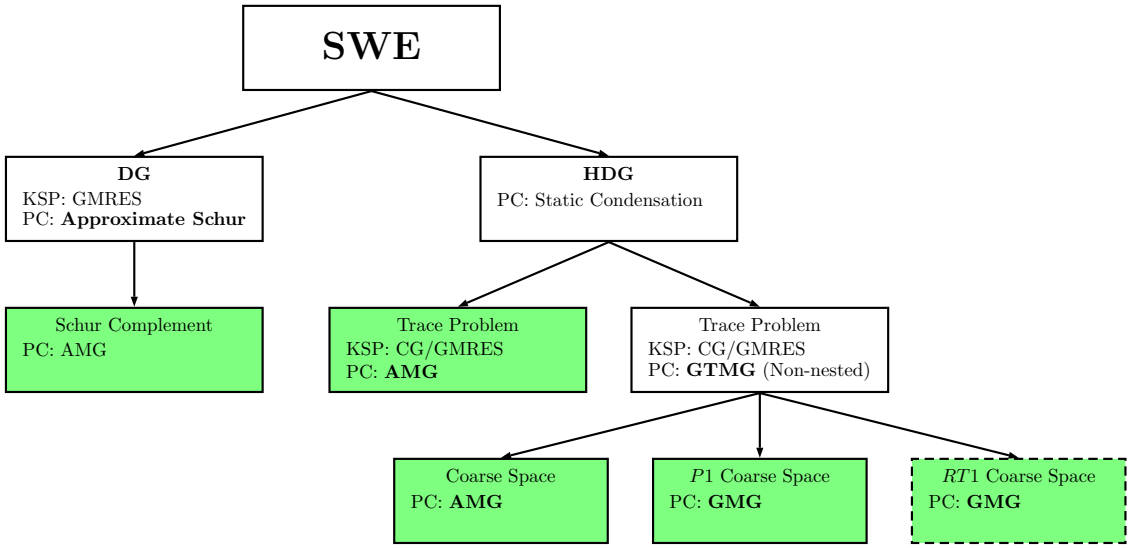

Figure 8-1: Tree of solver options

iterations required to reach this tolerance. For the approximate Schur complement method this is the number of outer iterations of the GMRES solver, but for HDG methods is the number of iterations to solve the trace problem.

Refer back to chapter 7 for the algorithms and PETSc options:

- DG + ApproxSchur (+ AMG):
  Section 7.6.1, Algorithm 7.4, Listing 7.8
- HDG + AMG:
  Section 7.6.2, Algorithm 7.5, Listing 7.9
- HDG + GTMG + AMG:
  Section 7.6.3, Algorithm 7.6, Listings 7.10 and 7.11
- HDG + GTMG + GMG:
  Section 7.6.3, Algorithm 7.6, Listings 7.10 and 7.12

8.1 Solvers

In this section we are interested in the matrix problem so we start with the linear shallow water equations. The linear SWE are an easier problem to initially test the solving strategies on than the non-linear SWE.

The parameters for the results in this section are summarised in table 8.3.

Problem	Linear equations + Constant bathymetry + upwind flux
Solvers	DG + Approx Schur HDG + AMG HDG + GTMG + AMG HDG + GTMG + GMG
Refinements	$r = 4, 5, 6, 7, \underline{8}$
Degrees	$p = 1, 3, 5$
Timestepper	Theta Method with $\theta = 0.5$

Table 8.3: Problem parameters for comparing all solvers

Refinement 8 is underlined as only HDG + AMG was run at this refinement for degree 5.

The problem is setup to include all of the solvers in figure 8-1, except for the $RT1$ coarse space solve. This provides a lot of data points, so rather than plot 12 overlapping lines, we separate the plots by degree. Here we assess each solver's performance compared to just the other solvers using the same degree. In figure 8-5 we can see the DG + ApproxSchur and HDG + GTMG + AMG solvers compared to other degrees in the same plot.

In order to compare the performance of all the different solvers we look at the run time for each of the methods at various degrees. We already know how the number of degrees of freedom vary with respect to both the degree and the refinements, this was summarised in table 8.2. In figures 8-2 to 8-4 we plot the runtime per timestep per DOF against the number of DOFs. The number of DOFs for the HDG method is the number of global DOFs excluding facet DOFs, so that a direct comparison can be made between the size of the DG and HDG systems.

The runtime per timestep per DOF or "work per DOF" is a quantity that we observe to be approximately constant as we increase the mesh refinement. As we increase the total number of DOFs by refining we increase the total amount of work, but only by an amount proportional to the number of DOFs. This demonstrates that the solvers are algorithmically optimal since the cost grows proportionally to the number of unknowns.

We start by looking at all the solvers for degree 1 in figure 8-2. There is not much difference in the work per DOF until we solve problems with more than 10^5 DOFs, at which point all of the non-nested multigrid preconditioned HDG solvers are significantly faster than the DG method. This speedup is not only due to the number of DOFs, but also that the HDG method requires fewer solver iterations, as we will see in section 8.2.

Figure 8-3 shows the work per DOF for all solvers at degree 3. Here the results are more spread out. At very small number of DOFs the ApproxSchur preconditioner is competitive with the GTMG preconditioner, but at large numbers of DOFs this is no longer the case. Beyond approximately 10^5 DOFs the HDG method is clearly beating the DG method.

Finally, the degree 5 results are shown in figure 8-4. Here the difference between the DG method and the HDG method is much more pronounced. Even at the smallest problem size HDG is faster than DG.

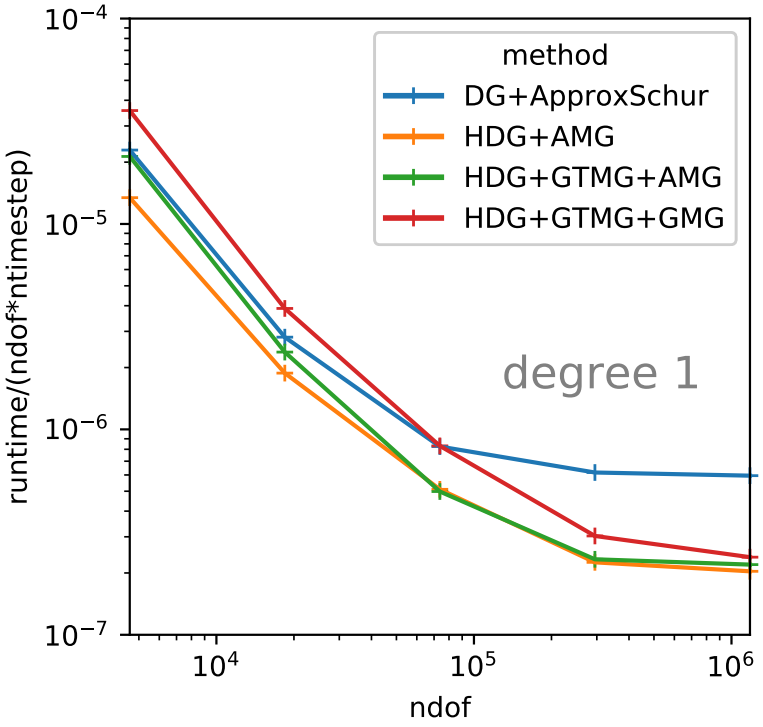

Figure 8-2: Runtime per DOF for a single timestep for different solvers and preconditioners for (H)DG1 discretisation of the linear SWE

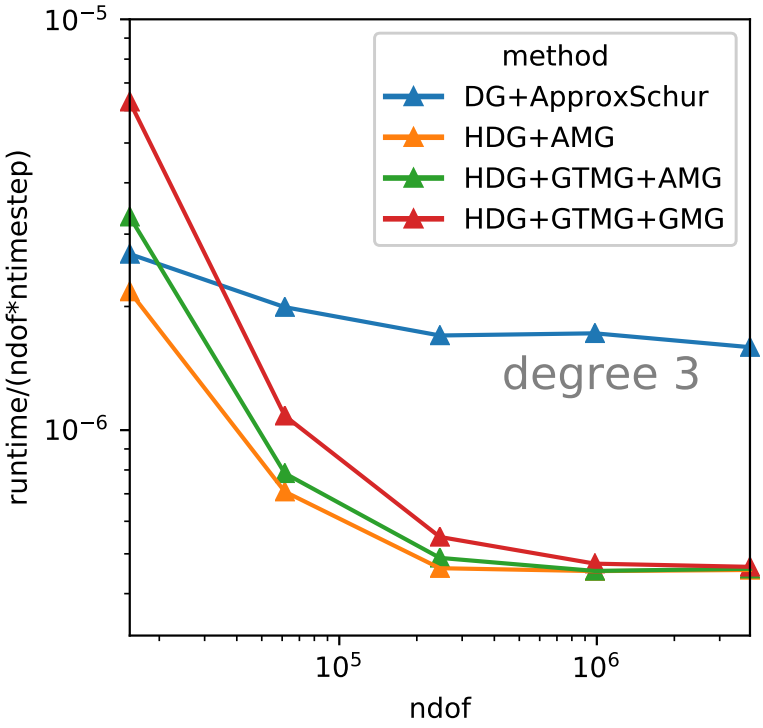

Figure 8-3: Runtime per DOF for a single timestep for different solvers and preconditioners for (H)DG3 discretisation of the linear SWE

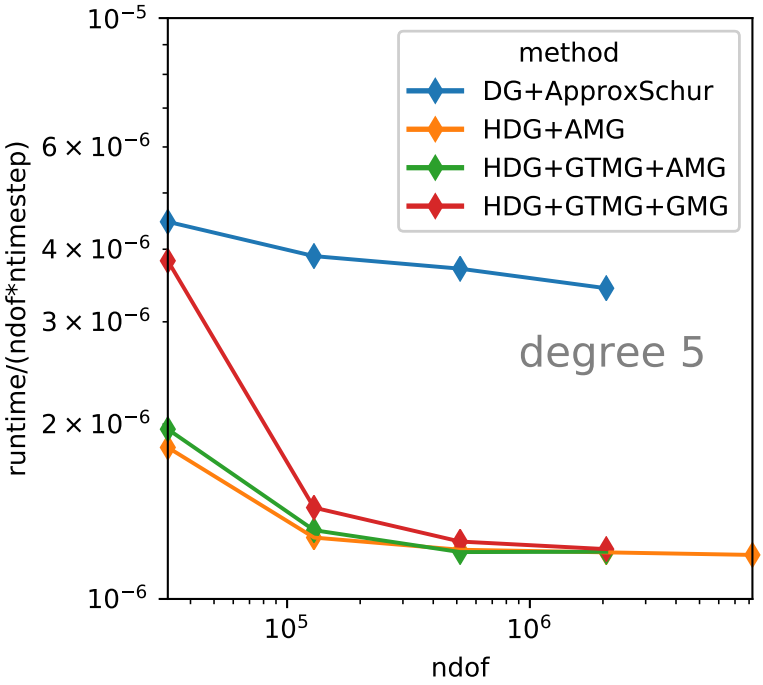

Figure 8-4: Runtime per DOF for a single timestep for different solvers and preconditioners for (H)DG5 discretisation of the linear SWE

From the three plots in figures 8-2 to 8-4 we can draw the following conclusions about our different solvers: The hybridised DG method *always* beats the approximate Schur complement preconditioned DG method, regardless of the choice of preconditioner, by a significant margin.

Geometric multigrid appears to struggle at very low refinement. This is perhaps due to the high setup cost of the mesh hierarchy, which for these results is not measured separately. This setup cost is not as apparent at higher order and is quickly absorbed into the work per DOF on more refined meshes. For a matrix free method, GMG is the only real choice. Since we do not use a method for constructing a mesh hierarchy directly on facets GTMG + GMG is currently the only way we can perform geometric multigrid. The current solver has not been tested as a matrix free solver and the PETSc options may need to be changed to allow for it to work in this way.

When considering AMG in isolation the benefit of GTMG + AMG over just using AMG on the trace space is not immediately apparent. But, it's worth noting that the AMG solver used in the PETSc solver library is a highly optimised piece of code. Furthermore, HDG + GTMG + AMG requires less memory than HDG + AMG as the assembled matrix will be smaller, HDG + GTMG + GMG can reduce the memory requirements further if a matrix free method is used. Our GTMG implementation, whilst certainly not inefficient and could be further optimised, is still very competitive. In fact, these plots suggest that the GTMG + GMG runtime is asymptotically approaching GTMG + AMG runtime and has comparable performance when solving problems with large numbers of DOFs.

Since the ApproxSchur preconditioner uses AMG as a preconditioner for the inner solve, it is most fair to compare the DG + ApproxSchur (+ AMG) solver to the HDG + GTMG + AMG solver as a means of assessing the effectiveness of the non-nested multigrid preconditioner. These are the two solvers that we compare in sections 8.2, 8.3 and 8.5.

8.2 Linear SWE

We consider the linearised SWE to start assessing the performance of the solver routines we have written. The parameters for the results in this section are summarised in table 8.4.

Problem	Linear equations + constant bathymetry + upwind flux
Solvers	DG + Approx Schur HDG + GTMG + AMG
Refinements	$r = 4, 5, 6, 7, \underline{8}$
Degrees	$p = 1, 3, 5$
Timestepper	Theta Method with $\theta = 0.5$

Table 8.4: Problem parameters for the linear SWE

Refinement 8 is underlined as only degrees 1 and 3 are simulated on this mesh size. Note that these results can be constructed from the data in section 8.1 and is figure 8-5 is plotted to compare DG and HDG for different degrees. We also present speedup in figure 8-6 and iteration counts in table 8.5 for these methods.

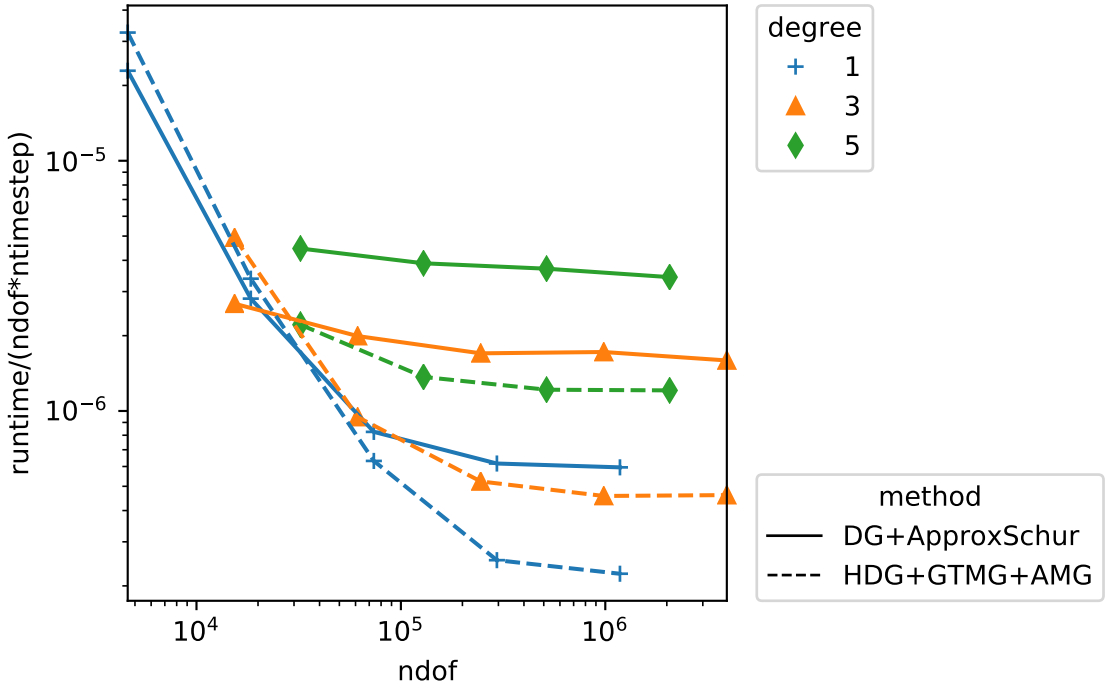

Figure 8-5: Runtime per DOF for a single timestep comparing different degree (H)DG discretisations of the linear SWE

Figure 8-5 shows that for coarse refinements, with a low number of DOFs, the work per DOF is much higher than at fine refinements. This is likely due to the additional amount of setup cost for the method for smaller problems.

In the highly refined region of the graph, the work per DOF has levelled out and is smaller for the non-nested HDG (HDG+GTMG+AMG) solver than for the approximate Schur complement preconditioned DG method for all degrees.

Figure 8-6 quantifies exactly how much faster the HDG method is over the DG method. The factor speedup measures how many times faster the HDG solver is than the DG solver

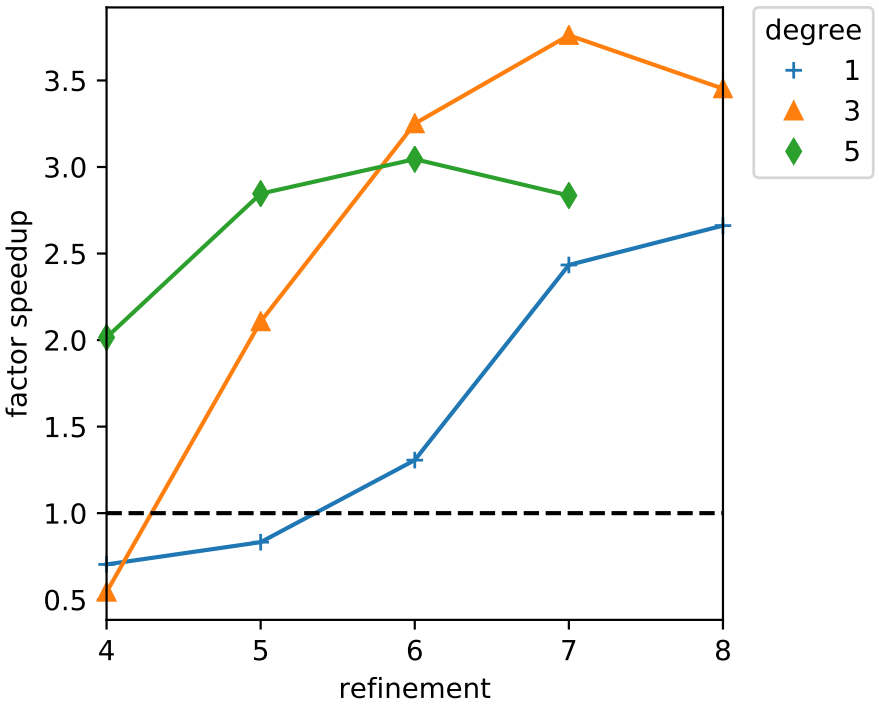

Figure 8-6: Relative speedup for the HDG method over the DG method for different mesh refinements and different degree discretisations of the linear SWE

at the same refinement for different degrees. The black dashed line on the plot indicates no speedup, a point sitting on this line would indicate that the HDG method took the same time as the DG method, and points below the line indicate the HDG method is slower than DG. Almost all data points are above this line, and importantly the points for higher refinement and higher degree are all above this line. At the highest refinement the HDG solver is between twice and four times faster for all degrees.

degree	$p=1$		$p=3$		$p=5$	
	DG	HDG	DG	HDG	DG	HDG
refinement						
$r=4$	14.1	8.1	15.5	8.0	17.3	8.0
$r=5$	13.8	8.1	14.9	7.0	16.5	8.0
$r=6$	13.0	8.0	14.0	7.0	15.9	8.0
$r=7$	12.9	8.0	14.0	7.0	15.0	8.0
$r=8$	12.0	8.0	13.0	7.0	-	-

Table 8.5: Average number of solver iterations required for the DG and HDG solvers (outlined in table 8.4) to converge when solving the linear problem

We are also interested in the number of iterations it takes for the solver to converge. One desired property for a correctly implemented multigrid preconditioner is mesh independent convergence, that is, the number of iterations remains constant, even when the mesh is refined. This is also referred to as h–robustness. Table 8.5 shows number of iterations required for the solver to reduce the residual to 10^{-8}, averaged over all of the timesteps.

For the DG method with the approximate Schur complement preconditioner, the average number of iterations increases with degree, but decreases with refinement. If this trend continues, for even bigger problems the number of iterations may decrease further. Table 8.5 shows that both DG and HDG are h–robust methods. However, in all cases the number of iterations required for the DG solver to converge is larger than the number of iterations required for the HDG solver to converge. The HDG solver with the non-nested multigrid solver and AMG on the coarse space consistently takes only 7 or 8 iterations to converge at every timestep, regardless of the degree or mesh refinement. At higher order the HDG solver requires only approximately *half* the number of iterations of the DG solver. This is exactly the property we want in our solver and is also known as p–robustness.

8.3 Non-linear SWE with bathymetry

We now consider the non-linear problem, which allows us to assess the solvers and timestepping routine in a more realistic setting. The parameters for the results in this section are summarised in table 8.6.

Problem	Non-linear equations + bathymetry + Lax-Friedrichs flux
Solvers	DG + Approx Schur HDG + GTMG + AMG
Refinements	$r = 4, 5, 6, \underline{7}, \mathbf{8}$
Degrees	$p = 1, 3, 5$
Timestepper	Theta Method with $\theta = 0.5$

Table 8.6: Problem parameters for the non-linear SWE

Refinement 7 is underlined as only degrees 1 and 3 with the non-hybridised solver were run due to time constraints. Refinement 8 is bold as it only gets run using the hybridised DG method and excludes degree 5.

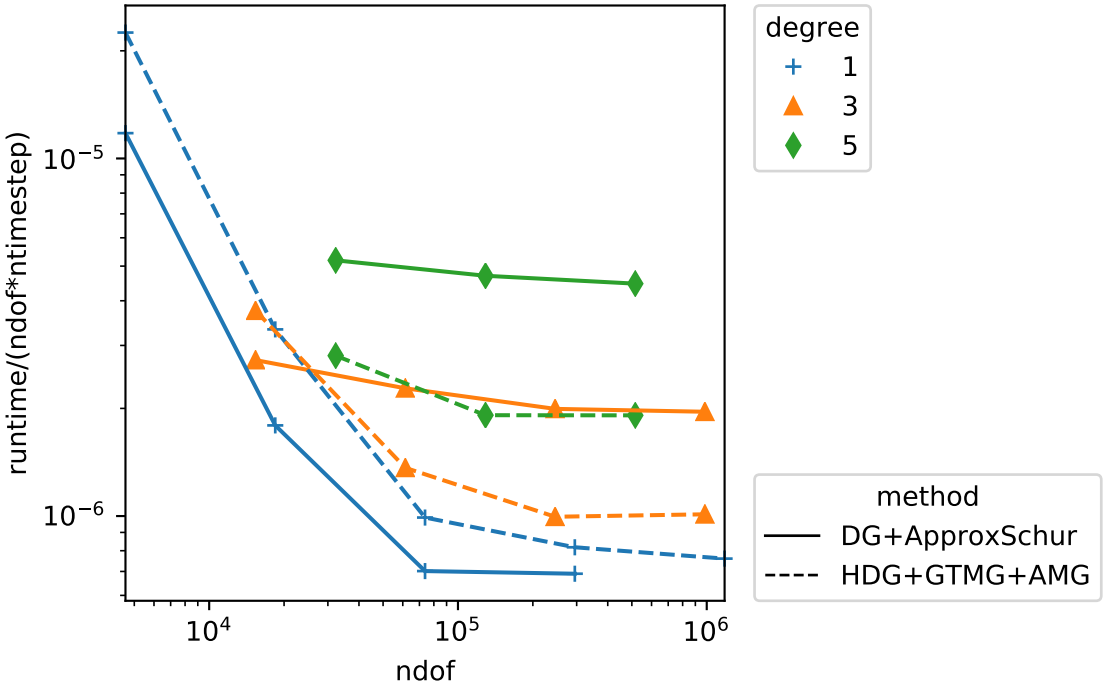

Figure 8-7: Runtime per DOF for a single timestep comparing different degree (H)DG discretisations of the non-linear SWE

We again use the "work per DOF" which is measured as the runtime per timestep per DOF, to assess the performance of each of the solvers. In figure 8-7 we see a similar pattern to the linear case shown in figure 8-5. For very small problems fewer than 5×10^4 DOFs the work per DOF is higher due to overheads, but above 5×10^5 DOFs the work per DOF levels out.

In this regime we see, just like in the linear case, that the HDG method is beating the DG method. This is very clear for degrees 3 and 5, is not the case for degree 1, where there is no speedup.

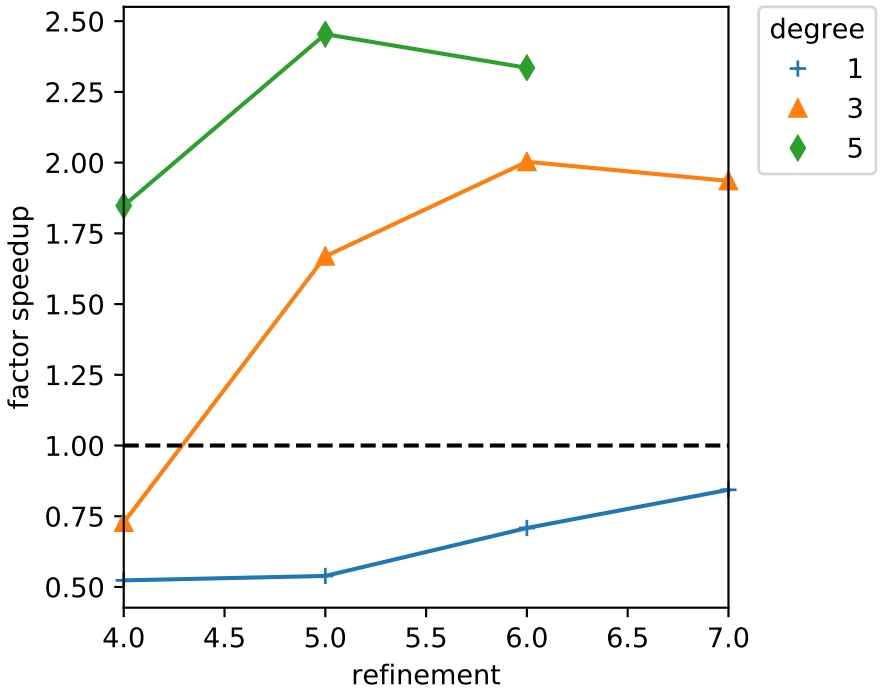

Figure 8-8: Relative speedup for the HDG method over the DG method for different mesh refinements and different degree discretisations of the non-linear SWE

This is confirmed in figure 8-8, where the line representing the degree 1 HDG method sits below the black dashed line. However, the degree 3 and degree 5 HDG methods still offer between a 1.5 and 2.5 times speedup over the DG method. It is also worth pointing out that in some cases we could run the HDG method for higher refinements than the DG method without the simulation timing out. This means that we cannot compute the speedup for these runs.

degree	$p = 1$		$p = 3$		$p = 5$	
	DG	HDG	DG	HDG	DG	HDG
refinement						
$r = 4$	15.9	10.1	18.0	8.9	21.0	8.8
$r = 5$	15.0	10.2	17.3	9.0	20.4	8.8
$r = 6$	14.1	10.1	16.3	9.0	19.4	9.0
$r = 7$	14.0	10.1	16.0	8.9	-	-
$r = 8$	-	10.0	-	-	-	-

Table 8.7: Average number of GMRES iterations required for the DG and HDG solvers (outlined in table 8.6) to converge when solving the non-linear problem

We also measure the number of iterations taken for GMRES to converge for the non-linear problem, the results are summarised in table 8.7. The number of iterations required for DG method with the approximate Schur complement preconditioner again increases with degree, but decreases with refinement. The number of GMRES iterations is slightly larger than the linear results in table 8.5, but this is less pronounced for the HDG method. Both the DG and HDG method are h–robust and table 8.7 shows this fact.

In all cases the number of iterations required for the DG solver to converge is larger than the number of iterations required for the HDG sovler to converge. The HDG solver with the non-nested multigrid solver and AMG on the coarse space consistently takes between 7 and 11 GMRES iterations to converge at every timestep, regardless of the degree or mesh refinement. This is significantly fewer iterations than are required for the DG + Approx Schur solver to converge.

8.4 Timestepping

In this section we look at the timestepping method we use to advance the solution forward in time. We want to include non-linear terms as this is a key part of the IMEX method discussed in section 4.2, so we revisit the non-linear problem section 8.3.

The parameters for the results in this section are summarised in table 8.8.

Problem	Non-linear equations + bathymetry + Lax-Friedrichs flux
Solver	DG + Approx Schur HDG + GTMG + AMG
Refinement	$r = 6$
Degree	$p = 3$
Timesteppers	Explicit Euler Heun SSPRK3 Theta Method $\theta = 0.5$ Theta Method $\theta = 0.55$ IMEX Theta (using IMEX framework) $\theta = 0.5$ IMEX Theta (using IMEX framework) $\theta = 0.55$ IMEX SSP2(3,2,2) IMEX ARS2(2,3,2)

Table 8.8: Problem parameters for comparing timesteppers

Rather than looking at different refinements and degrees we fix these and vary the timestepper. We have selected refinement 6 and degree 3 as they perform particularly well in the non-linear results in section 8.3. The DG and HDG methods are then timestepped using all the methods in the table.

We use two different forms of the theta method, with two different values of theta. One form uses the theta method described in section 3.1, the other uses the generic IMEX formulation described in section 4.2.1. Both forms are used to show that they produce the same result and to analyse the difference in performance.

The left-hand portion of the table we summarises the order of the method, the number of function evaluations and the number of solves required for each timestepping method, see section 4.2 for the Butcher tableau.

The theoretical order of the timesteppers is of little importance to the results, as we observe in the error column. Many low order methods actually produce higher accuracy results than theory would predict for suitably smooth solutions. In NWP the largest source of error does not come from the timestepper, and the tolerances for solvers are much looser, so it is not necessarily beneficial to use a high-order implicit timesteppers over a low-order one.

The evaluations are divided into $\mathcal{L}$–the number of linear function evaluations and $\mathcal{N}$–the number of non-linear function evaluations. The solves are broken down into M–the number of matrix solves using a DG mass matrix (section 3.1) and A–the number of system matrix solves, which are solves involving either A in the DG case (as defined in section 7.6.1, equation (7.15)), or $\widehat{A}$ in the HDG case (as defined in section 7.6.2 equation (7.16)).

The next columns show the total runtime and average iteration count (for solves involving A or $\widehat{A}$) in the DG and HDG cases. The error in the final column is only shown for the DG case, but since the DG and HDG problems are equivalent, the error will be

	ord	eval		solve		runtime		iterations		error
		$\mathcal{L}$	$\mathcal{N}$	M	A	DG	HDG	DG	HDG	DG
timestepper										
Explicit Euler	1	1	1	1	0	31.96	-	0	-	0.06254
IMEXSSP2(3,2,2)	2	3	2	1	3	327	158.7	17	8	7.361e-06
IMEXTheta θ =0.5	1	1	1	2	1	110.3	59.81	16.26	8.981	5.291e-06
Theta θ =0.5	1	1	1	0	1	102.9	52.52	16.26	8.986	5.291e-06
Heun	2	2	2	2	0	**35.27**	-	0	-	5.413e-08
SSPRK3	3	3	3	3	0	59.11	-	0	-	5.409e-08
IMEXARS2(2,3,2)	2	2	3	2	2	184.4	112	14	8.62	5.386e-08
IMEXTheta θ =0.55	1	1	1	2	1	117.3	**58.3**	17.93	8.883	5.376e-08
Theta θ =0.55	1	1	1	0	1	116.1	**56.56**	17.93	8.873	5.376e-08

Table 8.9: Runtime and iteration counts for different timesteppping methods using DG and HDG discretisations of the SWE

the same for both. Numerical error is discussed further in section 8.5.

Table 8.9 summarises the results for the parameters listed in table 8.8. We can compare the solvers for all of the timesteppers. The runtime of the HDG + GTMG + AMG solver is always smaller than the runtime of the DG + Approx Schur solver for the non-linear SWE with refinement 6 and degree 3. The iteration count is very consistently between 8 and 9 GMRES iterations for HDG, but the iterations for DG vary between 14 and 17. Not only is the iteration count smaller for HDG, it is also more consistent.

The rows have been sorted in order of decreasing error. At the top of the list is the explicit Euler timestepper. It is by far the fastest executing method, which is due to not having to solve using the full system matrix, just the DG mass matrix and only having to perform one function evaluation for each of the linear and non-linear functions. This means there are no runtimes or iteration counts in the HDG column, for this or any of the explicit timesteppers, as there is no need to hybridise since there are no coupled degrees of freedom in the mass solve. There are also no GMRES iterations in the DG column as a matrix equation only involving a DG mass matrix can be solved with a single pass of block SOR. The disadvantage of the explicit Euler method is it has the largest timestepping error, this is because it is only a first order timestepping method. Even with Δt chosen to be 1/20 the size of the implicit method's timestep the error is still orders of magnitude larger than any of the semi-implicit methods.

The next largest error is the IMEX SSP2(3,2,2) method, which also has the longest runtime. The longer runtime can be attributed to the method having three rather than two stages and performing a system matrix solve at all three stages. This timestepper seems to be a poor choice for the problem.

With $\theta = 0.5$ both the theta and IMEX theta methods have almost identical error and iteration counts. The IMEX formulation makes the method slower by a mere 8 seconds in this case. Both the $\theta = 0.5$ methods here have comparable error to the much slower IMEX SSP2(3,2,2) scheme.

The remaining methods in table 8.9 all have similar size error, we do not expect to get errors smaller than 10^{-8} as this is the tolerance we use for the solvers. First there are the remaining two explicit timesteppers, the order 2 Heun method and the strong stability preserving Runge-Kutta method of order 3. Both have small runtimes compared to all of the IMEX DG methods, since there is no system matrix solve, but produce an error smaller or comparable to these methods.

The IMEX ARS method is significantly slower than any other method with approximately 5×10^{-8} error. Like the SSP2(3,2,2) method, the ARS method has three stages, although only one stage requires a system matrix solve, and ARS requires three non-linear function evaluations, which further slow down the method. The theta method[2] with $\theta = 0.55$ is significantly faster and has the same magnitude error. Both the IMEX and non-IMEX formulation have very similar runtimes, suggesting that using the IMEX formulation causes only a very small overhead.

Importantly, the IMEX theta method using the HDG solver only 60% slower than Heun's explicit timestepper. The two methods have a similarly small error, making a direct comparison fair. These results have been put in bold to highlight them in table 8.9. Note that Heun consists of two explicit steps, hence two mass matrix solves and IMEX theta has two steps, consisting of two mass matrix solves and one solve for the hybridised system. The IMEX theta doesn't have to perform as many non-linear function evaluations, this combined with the efficiency of the HDG + GMTG + AMG solver makes the off-centred IMEX theta timestepper efficient. Our off-centred IMEX theta timestepper is not quite as competitive when compared to the explicit Heun method, but is still an impressive speedup over traditional DG methods.

[2] sometimes called an off-centred theta method

8.5 Error

We mentioned at the start of the chapter how important it is to ensure that our numerical solutions agree with known analytic solutions. This can be taken a step further, if we measure the error at each different grid refinement we can determine the rate of convergence for a specific degree. By calculating the error, we can check the correctness of the code we have written.

The parameters for the results in this section are summarised in table 8.10. We note the solvers we use, but the specific solver is not relevant, since the DG and HDG methods generate identical solutions and will have the same error.

Problem	Non-linear equations + bathymetry + Lax-Friedrichs flux
(Solvers)	(DG + Approx Schur) (HDG + GTMG + AMG)
Refinement	$r = 4, 5, 6, 7$
Degrees	$p = 1, 3, 5$
Timesteppers	ARS2(2,3,2)

Table 8.10: Problem parameters for rate of convergence

We have chosen the (IMEX) ARS2(2,3,2) timestepping method here which is a second order timestepping method, that we have seen to be very accurate in section 8.4. Some of the error that we measure will come from the choice of timestepping method, rather than the spatial discretisation. We choose ARS2(2,3,2) as it ensures we minimise the amount of error we make due to time discretisation, and the measured error predominantly comes from the DG method.

For problems like the advection equation that we saw in section 3.2 it can be proven that for suitably smooth solutions the rate of convergence is guaranteed to be $O(h^{p+1/2})$ [33], where p is the polynomial degree. It has been observed that the DG method can actually exceed this rate of convergence and attain $O(h^{p+1})$ convergence [18].

To measure the error we calculate

$$\varepsilon = \left(\int_\Omega (\boldsymbol{q}_h - \boldsymbol{q}_{\text{true}}) \cdot (\boldsymbol{q}_h - \boldsymbol{q}_{\text{true}}) \, dx \right)^{1/2},$$

where $\boldsymbol{q}_h$ is the solution to the non-linear SWE in equation (3.30) and $\boldsymbol{q}_{\text{true}}$ is the rotating vortex defined in equations (2.11) to (2.13).

We show, in figure 8-9, that as we refine the mesh our solution does become more accurate and we plot the results.

A rate of convergence somewhere between $O(h^{p+1/2})$ and $O(h^{p+1})$ is observed for all degrees in figure 8-9. The black dashed lines show the gradient for perfect $O(h^e)$ convergence for $e = 1.5, 2, 3.5, 4, 5.5, 6$. The rate of convergence is given by how closely the gradients of the lines match, not how close the plot lines get to the black dashed lines. This figure is identical for the DG and HDG methods, which we should expect. We saw in section 4.1.1 that when we eliminate the trace variable the DG and HDG problems are equivalent. Therefore, when we solve them numerically we expect the error to be the same for both methods.

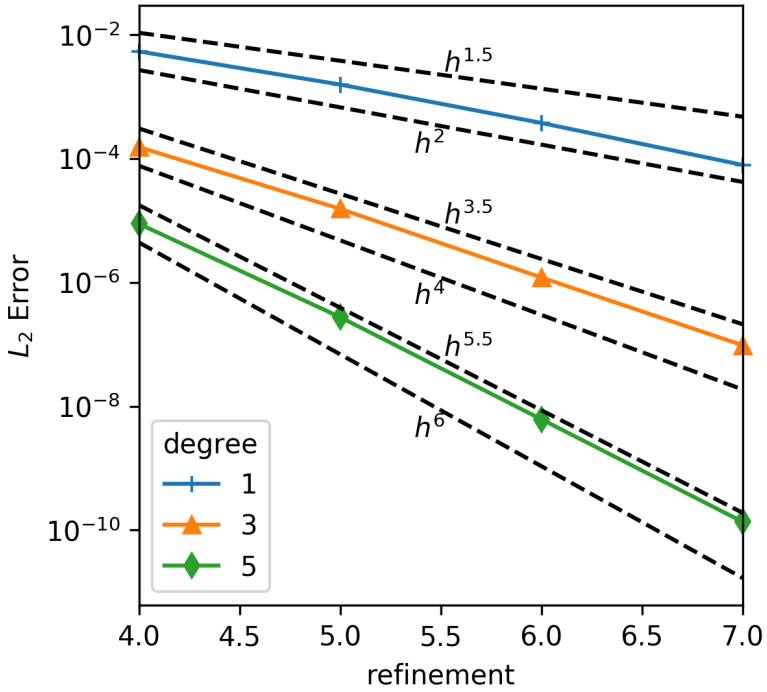

Figure 8-9: L_2 error for different mesh refinements using DG and HDG discretisations of the non-linear SWE showing the rate of convergence for different degree methods

8.6 Scaling

When assessing the parallel performance of code there are two standard tests we perform:

- **Strong scaling:** The total problem size stays fixed, as the number of processors[3] increases. For problems in numerical weather prediction often the goal is to minimise the time to obtain a solution in order to perform model runs in an operational time frame. For the Met Office an operational runtime is around 20 minutes [4]. One way of reducing the runtime is to use more processors.
- **Weak scaling:** The total problem size increases proportionally with the number of processors, keeping the amount of work per processor fixed. In NWP high resolution model runs have too many DOFs to fit into RAM on a single node so the only way to complete a big model run is to spread it over multiple nodes.

In the introduction to chapter 3 we claimed that DG was an ideal method for implementing in parallel due to the local data layout, which cuts down on parallel communication, and higher arithmetic intensity at high order, which suits modern architectures.

In this section we will demonstrate that this claim is true and that the DG method can be implemented efficiently in parallel. Furthermore, the solvers we have implemented remain efficient even when run in parallel across hundreds of processors.

The Firedrake documentation [2] suggests having at least 50 000 DOFs per core in order to see good parallel scaling. In the case of the University of Bath HPC facility, *Balena*, this means 800 000 DOFs per node. We can use the values summarised in table 8.2, which gives the total number of DOFs for the problem at a given refinement and degree, to guide the choice of problem size.

8.6.1 Strong scaling

The parameters for the strong scaling results are summarised in table 8.11.

Problem	Linear equations + Constant bathymetry + upwind flux
Solvers	HDG + GTMG + AMG
Refinement	$r = 7$ $r = \underline{10}$
Degrees	$p = 1, 3, 5$ $p = \underline{1}$
Timesteppers	Theta Method $\theta = 0.5$
Processors	16, 32, 64, 128, 256

Table 8.11: Problem parameters for strong scaling

Refinement 10 and degree 1 are underlined as they correspond to a second strong scaling run.

We start the strong scaling test from 16 processors which is one full node. This avoids having to solve a problem with a very large number of DOFs on a single core, which would take an unreasonably long time to complete. We are more interested in how our solvers

[3] In this discussion we use processor to mean CPU core and the terms core and processor are used interchangeably. In reality a node contains a number of 'sockets' and each socket contains a 'CPU' chip (sometimes referred to as a processor) which itself contains multiple 'cores'. Using core or processor to refer to one of these compute units simplifies the discussion.

scale to very large numbers of processors in a HPC environment than how well it scales on a single node. Starting from the initial run with 16 processors, for each successive run the number of processors is doubled until we reach 256.

A refinement of 7 is chosen as it is a good compromise between getting the biggest problems to run in a reasonable time on one node and having enough work per core when we run on 256 cores.

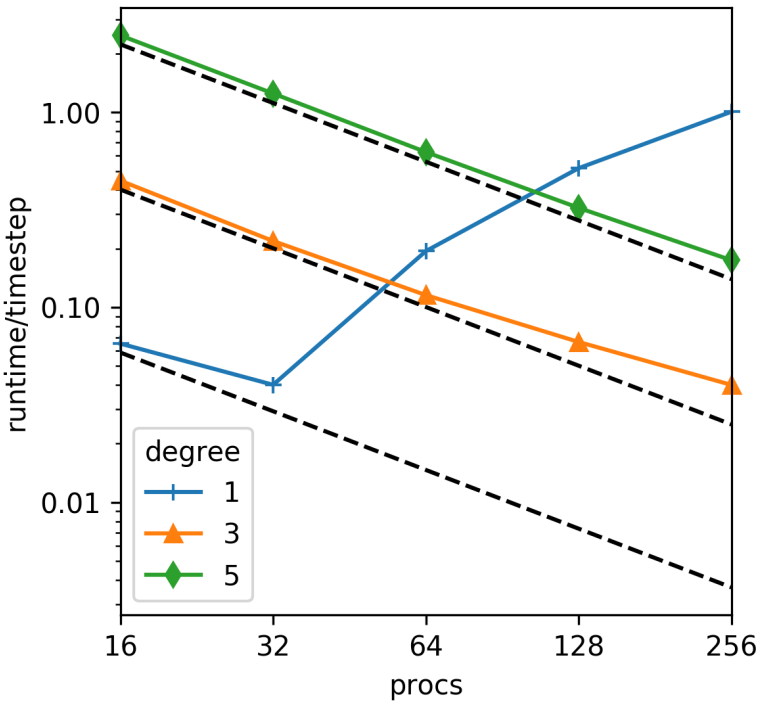

Figure 8-10: Strong scaling run showing the runtime per timestep on an increasing number of processors for a fixed problem size and different degree HDG discretisations of the SWE

On 256 processors the refinement 7 problem has approximately 10^4 DOFs per processor for a degree 1 method, 4×10^4 DOFs per processor for a degree 3 and 8×10^4 DOFs per processor for degree 5. This falls very short of 50 000 DOFs per core, but as figure 8-10 shows, this does not seem to have affected the scaling performance (certainly not for degree 3 and 5). The black dashed line represents perfect scaling for each degree. Perfect scaling would mean that when we double the number of processes the total runtime halves. Figure 8-10 plots the runtime per timestep, which also halves with perfect scaling as the problem size (and hence total number of timesteps) remains fixed.

For degrees 3 and 5 we observe almost perfect strong scaling up to 256 cores. This is a great result as we are primarily interested in higher order DG methods and figure 8-10 shows that the higher the order of the method the better the scaling, especially at larger processor counts.

The degree 1 problem has some issues with strong scaling. When we look at the total number of degrees of freedom in the problem we can see that there are only approximately 3×10^5 DOFs, which for 64 cores (4 nodes, where the method stops strong scaling) this is only approximately 5 000 DOFs per core. This is unlikely to be a sufficiently high load to keep the CPU busy while communication between nodes takes place. We conjecture that the degree 1 problem spends much of its time waiting for communication.

In order to rectify this issue we perform a strong scaling run for degree 1 that uses

refinement 10 and extend the allowed runtime. A refinement 10 problem has almost 2×10^7 DOFs, so even when we run on 256 processors, this has almost 75 000 DOFs per core.

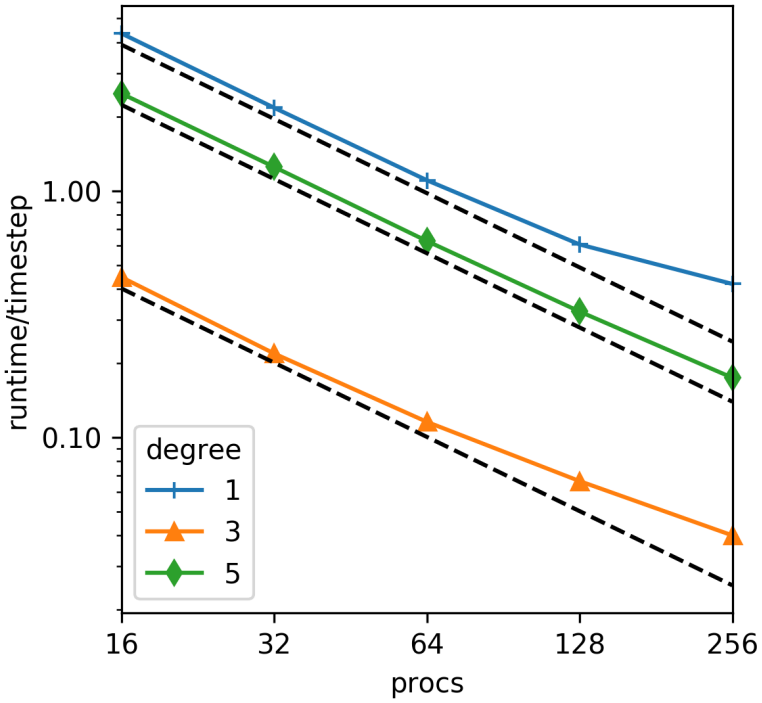

Figure 8-11: Strong scaling run like figure 8-10 but using a larger refinement for the degree 1 HDG discretisation of the SWE

Figure 8-11 shows the additional degree 1, refinement 10 strong scaling run alongside the degree 3 and 5 refinement 7 strong scaling runs. We see that the degree 1 HDG solver is now in the strong scaling regime and performs similarly to the lower refinement runs using higher degrees. Looking the scaling in figure 8-11 we clearly see that higher order DG performs better than low order DG in this strong scaling test.

8.6.2 Weak scaling

The parameters for the weak scaling results are summarised in table 8.12.

Problem	Linear equations + Constant bathymetry + upwind flux
Solvers	HDG + GTMG + AMG
Refinement	$r = 4, 5, 6, 7, 8$ $r = \underline{7}, \underline{8}, \underline{9}, \underline{10}, \underline{11}$
Degrees	$p = 1, 3, 5$ $p = \underline{1}$
Timesteppers	Theta Method $\theta = 0.5$
Processors	1, 4, 16, 64, 256

Table 8.12: Problem parameters for weak scaling

The second line of refinements is underlined as is degree 1 as these correspond to a second weak scaling run.

For weak scaling we need the problem size to scale proportionally to number of processors. Since our problem is in 2D, refining the mesh once halves cell diameter and this quadruples the number of DOFs. This means that for each successive run we quadruple the number of processors until we hit 256. For this reason we start weak scaling runs on one processor, not one node as we did for strong scaling. The refinement size is chosen to keep number of DOFs per core constant and close to 50 000 DOFs per core for degree 5, whilst running in a reasonable time.

Recall from section 4.2 that the CFL condition imposes a restriction on the timestep size we use and this is dependent on the grid spacing. If we halve the cell diameter, we must also halve the timestep size and must take twice as many timesteps to cover the same time period. This means refining the mesh once corresponds to an 8 times total computational workload increase.

We look at the average time it takes to perform one timestep, not the total runtime as we expect this to scale weakly. The runtime per timestep for each processor count is plotted in figure 8-12.

We expect the lines in this plot to be constant because, if the work grows linearly with the problem size n then the work per processor stays constant. The grey region in figure 8-12 represents weak scaling within a single node where the number of processors is less than 16, and the rest of the plot is weak scaling between multiple nodes.

On 256 processors the largest problem size is approximately 1.2×10^6 DOFs for the degree 1 method, 4×10^6 DOFs for degree 3 and 8×10^6 DOFs for degree 5.

For degrees 3 and 5 we observe almost perfect weak scaling up to 256 cores. In both cases the lines are almost perfectly constant, note that the scale on the y–axis is *not* logarithmic as it was in the strong scaling plots. This means that we can theoretically keep using even more processors to solve bigger and bigger problems and the time to advance one timestep will remain constant.

The degree 1 problem has some issues with weak scaling since, we conjecture that the problem size is not large enough and the time we see is dominated by the inter-node communication time. On 256 processors a single timestep takes more than five times longer than on 16 processors.

We repeat the weak scaling run with a sequence of more highly refined meshes to create problems with higher DOF counts. The largest problem for degree 1 is now 7.5×10^7 DOFs.

Figure 8-13 shows the results running degree 1 with higher refinements, compared

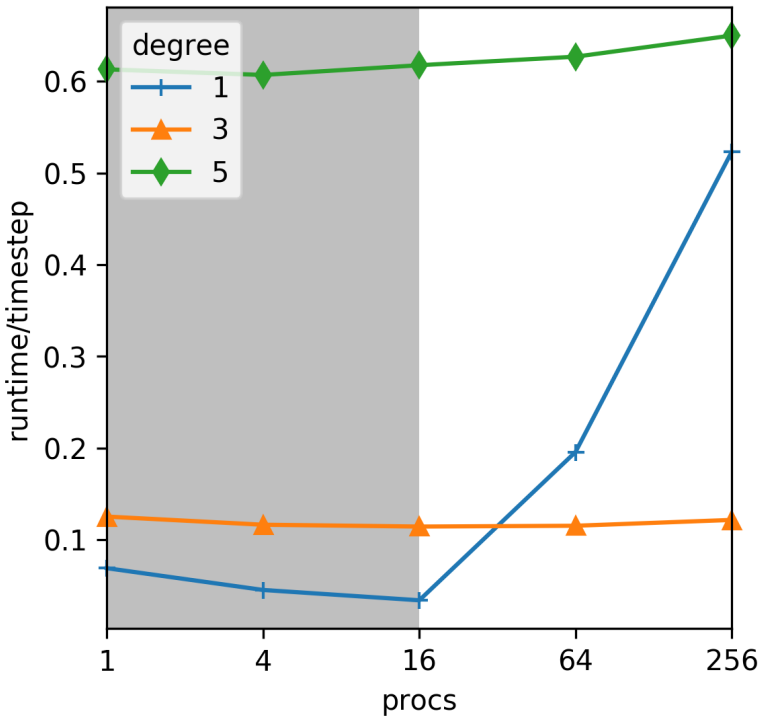

Figure 8-12: Weak scaling run showing the runtime per timestep on an increasing number of processors for a growing problem size and different degree HDG discretisations of the SWE

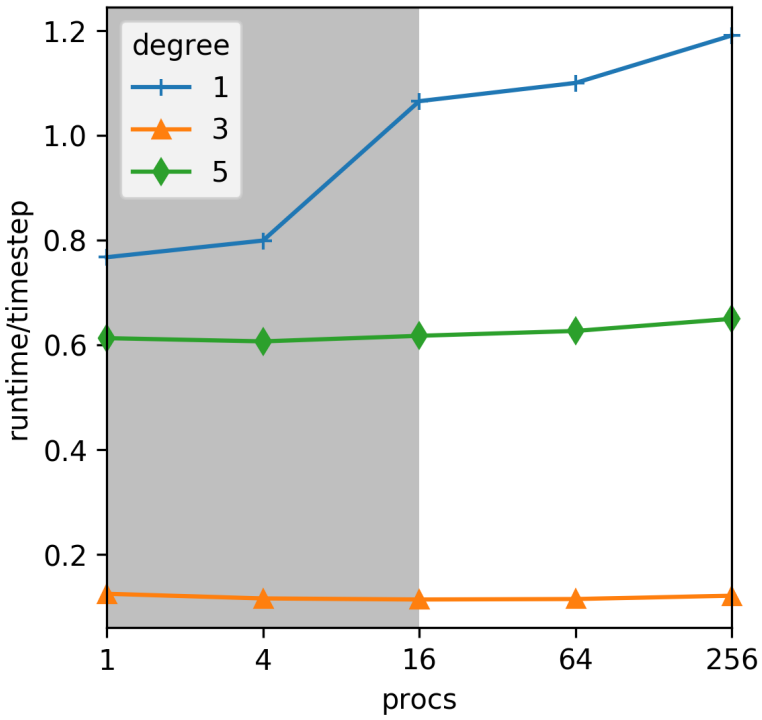

Figure 8-13: Weak scaling run like figure 8-12 but using a larger refinements for the degree 1 HDG discretisation of the SWE

to the degree 3 and 5 runs performed previously. Although the degree 1 line is not constant, this is better weak scaling than what was observed in figure 8-12. Inside the grey region, which corresponds to a single node, using anything fewer than all the cores increases the memory bandwidth available to each core. This accounts for the notably better performance on 1 and 4 cores. Beyond one node the degree 1 line is much flatter than before and the runtime per timestep is almost constant.

Another desirable property of solvers is that as the problem size increase, the number of iterations the solver takes to reach a fixed tolerance remains constant. This is known as h–robustness. When performing a weak scaling test this is important, because if the solver it is not robust, it will weak scale poorly. This could explain the poor scaling for degree 1, except table 8.5 shows that even at degree 1 the HDG solver is h–robust.

So for small problems degree 1 has good intra-node weak scaling and for large problems degree 1 has good inter-node scaling. For higher order DG this is less of a concern as both degree 3 and degree 5 methods exhibit excellent weak scaling both within and beyond one node.

CHAPTER 9

CONCLUSIONS

Numerical weather prediction requires fast solvers for the equations of fluid dynamics. This motivates the work we have done in this thesis, where we have shown the viability of using the DG method with semi-implicit timesteppers to numerically solve the shallow water equations. Due to the separation of fast and slow waves in atmospheric dynamics, semi-implicit timesteppers are highly efficient for problems in NWP, since the implicit treatment of fast waves avoids strict timestep size constraints. We address the issue that DG methods are generally regarded as unsuitable for semi-implicit timestepping methods, such as IMEX methods, since their current treatment requires expensive solves at every timestep. The key issue with their current treatment is that an effective Schur complement preconditioner cannot easily be constructed for DG.

Discontinuous Galerkin discretisations have many advantageous properties, since they can be used for arbitrary domains and support a structured local data layout, which is especially important on modern HPC systems. Furthermore, for suitably smooth solutions, higher order approximations can be efficient since the rate of convergence is proportional to the polynomial degree, which we demonstrate in section 8.5.

The coupled degrees of freedom in a DG discretisation make the Schur complement dense, so the solves required for IMEX timestepping are computationally expensive. We have used hybridisation techniques to reduce the number of coupled degrees of freedom by introducing an additional Lagrange multiplier situated on the facets of the underlying mesh. This new variable eliminates off-block diagonal entries in the DG system matrix and allows a sparse Schur complement to be formed. It is then possible to transform the SWE problem into a problem defined on the facets of the mesh, which requires solving a sparse system with far fewer degrees of freedom.

For the shallow water equations we constructed a preconditioner based on the hybridised discontinuous Galerkin method combined with a non-nested multigrid solver on the facets, which is suitable for higher order DG discretisations. These preconditioned HDG methods have been effectively combined with semi-implicit timestepping. And all that has been achieved whilst retaining efficiency and scalability of the method. Our preconditioner was compared to a more conventional approximate Schur complement factorisation approach to preconditioning.

We showed how we have used the Firedrake software framework, a modern code generation tool for finite element problems, to create an efficient implementation of these solvers and preconditioners, and presented the encouraging results of our numerical experiments.

This work shows that a suitable DG method, solver and preconditioner, implemented using modern software frameworks and run on the latest HPC architectures can not only perform competitively with other methods, but may well be suitable for the next generation of models for numerical weather prediction and atmospheric modelling.

9.1 Mathematical results

In the last 50 years interest in the DG method has continued to grow and, given the attractive properties of the method, we expect the trend to continue. One of the key reasons for interest in DG is its use on modern computers that consist of many processors working in parallel.

We expect that the current trend in computer architecture to continue, this means we must continue to efficiently use processors that can perform calculations far faster than they can access values from memory. With the breakdown of Dennard scaling[1] [25, 13] in 2006 we expect newer CPUs to become more powerful by increasing the number of cores rather than clock speed. This, in turn, requires parallel algorithms that can take full advantage of a processor's parallelism as well its speed. We must look at the numerical methods that complement these architectures, such as DG.

To ameliorate some of the expense of performing DG solves using a (dense) Schur complement factorisation, we investigated hybridisation. Hybridisation is a powerful tool because of the way in which it simplifies the system matrix of the associated problem. The purpose of using hybridisation is obtain a sparse Schur complement factorisation by eliminating coupled degrees of freedom. The end product of hybridisation is a smaller *sparse* matrix solve, that has an associated problem based on the facets of the mesh used with the DG method. To solve this system we wanted to use an efficient solver, ideally multigrid.

Multigrid is an efficient method due to $O(n)$ algorithmic complexity. It is an incredibly powerful tool, because of the recursive strategy that repeatedly reduces the size of the problem, whilst reducing the error at every scale. Reducing the size of the problem requires coarsening the underlying mesh, but for a problem on the skeleton mesh that arises from hybridisation it is not clear how to effectively coarsen[2].

Non-nested multigrid is non-standard multigrid technique that allows us to transport a problem from one space to another. In this specific case we want to move away from a problem on the skeleton mesh to something more familiar. By using a non-nested multigrid we moved the hybridised problem from the trace function space to a more familiar P^1 space, where we already have the theory and necessary tools to execute the multigrid algorithm.

DG methods, hybridisation and multigrid are all incredibly powerful tools, even in isolation, but through our investigation into each of them, we have shown it is possible to combine them into an efficient elliptic solver. Whilst this is a complicated nested solver, these layers are all necessary to tackle the size of problems that are encountered in NWP.

9.2 Numerical results

Numerical results show that our solver is not only capable of handling large problems, but also remains efficient. The results show that the solver is sufficiently efficient for a

[1] essentially that processor speed doubles every 18 months

[2] Wildey et al. [56] present an alternative approach in their paper.

semi-implicit timestepper using HDG and non-nested multigrid it is only 60% slower than an explicit timestepper employing a DG discretisation. The HDG method also executes approximately twice as fast as the equivalent DG method for the same timestepper.

In chapter 7 we described how we implemented the solvers for the DG method using an approximate Schur complement preconditioner and for an HDG method using a non-nested multigrid preconditioner. This used a combination of different pieces of modern software (Firedrake, UFL, Slate, PETSc), which focused on a separation of concerns and allowed us to express problems at a high level of abstraction whilst generating code that remains efficient.

This allowed us to experiment with a wide range of advanced solvers. Composability meant that different factorisations, solvers and preconditioners could be combined into one solver.

The numerical results we obtained are very encouraging. The initial investigation using a simplified linear problem and constant bathymetry demonstrated that the nested combination of solvers, that is hybridisation combined with non-nested multigrid (HDG + GMTG + AMG) worked correctly. Not only was the solver efficient, but it outperformed the more conventional approximate Schur complement factorisation method (DG + ApproxSchur).

We also studied a non-linear problem that is more like a real world problem. This formulation included all of the non-linear terms of the SWE as well as a spatially varying bathymetry. The solver performed just as well with the more complicated setup, even in cases where the DG + ApproxSchur struggled.

The important result we obtain is found in section 8.4, where we combined the two solution strategies with various different timesteppers. One of the main criticisms of DG methods in general is that it leads to a dense Schur complement created by the numerical fluxes. That, combined with the extra DOFs required to represent a discontinuous solution, makes the resulting system matrix too expensive to solve with. If DG matrices are too expensive to solve with then it limits the choice of timesteppers to explicit methods which only require DG mass matrix solves and timesteps are severely limited by the CFL condition. However, by using hybridisation and non-nested multigrid we have significantly improved the time it takes to perform a system matrix solve. In section 8.4 we showed that, with the right choice of IMEX timestepper, it is now possible to perform semi-implicit timestepping that performs only 60% slower than an explicit method with similar accuracy.

The solver is also shown to scale well, particularly for higher order DG methods. In section 8.6 we performed standard strong scaling and weak scaling and achieved near perfect scaling. This is important for tackling bigger problems that are encountered solving the size of problem that is encountered in real world application like NWP.

9.3 Future work

There are several ways of extending the work presented in this thesis.

In the short term, one of the first steps would be to finish implementing the code to allow GMG to be used on the non-nested coarse space when the numerical flux is *vector valued*. We have seen an example of this when we studied the Lax-Friedrich numerical flux. This code must ensure that the options for `pc_patch` (appendix A.4) get passed through the many nested layers of PETSc solver options correctly. At which point `firedrake.GTMGPC` could be integrated into the Firedrake project as a general tool for performing non-nested multigrid on other problems.

We have yet to try and utilise Firedrake's automatic sum factorisation, which gives a reduction in complexity and further speed up for high order DG methods. We believe that sum factorisation will speed up the method further since it increases the arithmetic intensity of the algorithm. An arithmetically intense algorithm is better suited to modern architectures which are capable of performing arithmetic operations faster than memory operations. Sum factorisation relies on using quadrilateral cells in the mesh, so the basis has tensor product structure. After ensuring the mesh is formed of quadrilaterals and turning on the sum factorisation feature everything else is handled by the form compiler automatically.

Solving very large problems is only possible by applying matrix free techniques as even the biggest HPC systems would not be able to store such large system matrices in RAM. Matrix free methods never store the assembled matrix, instead it's action is performed by a function, eliminating the need to store the full system matrix at all. We could replace our current matrix based preconditioning by an approximate matrix-free iterative inversion described by Bastian et al. [12], which could lead to further speedups for high polynomial degrees. The details of precisely how to perform hybridisation in a matrix free way need to be carefully worked out. We would no longer be able to use AMG as a solver as this explicitly requires the assembled matrix to work, we would need to rely on GMG. In section 8.1 we saw GMG was just as fast as AMG for large problems.

In this research we did not implement the shallow water equations on the surface of a sphere. In order to test our solvers are suitable for global atmospheric modelling, we would have to look at a spherical formulation. Working in a curved geometry provides additional challenges, for instance a mesh needs to be chosen so that degrees of freedom aren't clustered around the poles. Care also needs to be taken with the difference between surface normals and cell normals, which are a key part of the numerical flux definition. Once we have a spherical formulation we can begin to look at standardised test cases, for example the seven Williamson tests for SWE on a sphere [57].

Implementing these standard tests and applying our solvers would allow us to compare to other solvers already in use, not just for DG methods but for conforming finite elements, finite differences and other spatial discretisations also. Currently finite difference methods are in use by the UK Met Office[3] and spectral methods are used by ECMWF. A spherical formulation would allow for a realistic comparison of conforming finite element methods and DG.

As a much longer term goal, we would consider implementing the three dimensional Navier-Stokes equations. To design an efficient solver for higher-order DG methods for atmospheric modelling we must apply what we have found to the Navier-Stokes equations, a hyperbolic and fully non-linear system of equations which describe the flow of compressible fluids [24] in terms of the state vector

$$\boldsymbol{q} = (\boldsymbol{v}, \theta, \rho, \pi)^\top.$$

Here, $\boldsymbol{v}$ is the velocity, θ is the potential temperature, ρ the fluid density and π the pressure. Improving the solution time for the Navier-Stokes equations is of interest to the numerical weather prediction community, since these are the underlying equations used in forecasting the weather. Collecting the unknowns into a vector we can write the Navier-Stokes equations as

$$\frac{\partial \boldsymbol{q}}{\partial t} = \mathcal{N}(\boldsymbol{q}) + \mathcal{L}\boldsymbol{q},$$

[3] although the Met Office plan to move over to finite element framework in the near future

which we can treat in a similar way to the SWE. We can discretise time now using the theta method (theta denoted by α to avoid conflict with potential temperature):

$$\boldsymbol{q}^{(n+1)} - \alpha\Delta t\mathcal{L}\boldsymbol{q}^{(n+1)} = \boldsymbol{q}^{(n)} - (\alpha-1)\Delta t\mathcal{L}\boldsymbol{q}^{(n)} + \mathcal{N}(\boldsymbol{q}^{(n)}).$$

Some thought needs to go into finding a linearisation which captures the fast modes of this system. A hybridised formulation of Navier-Stokes has been proposed by Peraire et al. [42], but does not discuss efficient solvers. Using this formulation would allow us to perform semi-implicit timestepping, just as we saw for the SWE. The SWEs are a good model system for the Navier-Stokes equations as they show the same separation of wave speeds.

We have already shown that solvers using HDG combined with non-nested multigrid are significantly faster than traditional DG methods when performing implicit timestepping for the shallow water equations. If we show the same is true for Navier-Stokes, this would be a huge step in establishing the viability of high order DG methods for numerical weather prediction.

APPENDIX A

APPENDIX

A.1 Equivalent formulations of upwind flux

We saw the upwind flux first in equation (3.12), where we wrote the flux for advection as

$$(\boldsymbol{\beta}\phi_h)^* := \begin{cases} \boldsymbol{\beta}\phi_h^- & \text{if } n^- \cdot \boldsymbol{\beta} > 0 \\ \boldsymbol{\beta}\phi_h^+ & \text{if } n^- \cdot \boldsymbol{\beta} \leq 0. \end{cases}$$

We claim this is equivalent to the formulation in terms of jumps and averages given in equation (3.13), which is

$$(\boldsymbol{\beta}\phi_h)^* := \{\{\boldsymbol{\beta}\phi_h\}\} + \frac{1}{2}\ |\boldsymbol{\beta}\cdot\boldsymbol{n}|\ [\![\phi_h]\!]. \tag{A.1}$$

To see that the two forms are equivalent we start by considering the vector

$$\boldsymbol{v}^+ = \frac{1}{2}(\boldsymbol{\beta} + |\boldsymbol{\beta}\cdot\boldsymbol{n}|\cdot\boldsymbol{n}^+).$$

By itself this quantity isn't particularly useful, but note that on an edge E between the cells K_+ and K_-:

$$\boldsymbol{v}^+\cdot\boldsymbol{n}^+ = \begin{cases} \boldsymbol{\beta}\cdot\boldsymbol{n}^+ & \text{if } \boldsymbol{\beta}\cdot\boldsymbol{n}^+ > 0 \\ 0 & \text{otherwise,} \end{cases} \quad \text{and} \quad \boldsymbol{v}^+\cdot\boldsymbol{n}^- = \begin{cases} 0 & \text{if } \boldsymbol{\beta}\cdot\boldsymbol{n}^+ > 0 \\ \boldsymbol{\beta}\cdot\boldsymbol{n}^- & \text{otherwise} \end{cases}$$

So if the advection vector points into the cell K^-, figure A-1a and d,

$$\int_E \boldsymbol{v}^+\cdot\boldsymbol{n}^+\phi_h^+\psi^+\ ds = \int_E \boldsymbol{\beta}\cdot\boldsymbol{n}^+\phi_h^+\psi^+\ ds,$$

which is the correct flux contribution using the quantity ϕ_h^+ to the cell K^+. And

$$\int_E \boldsymbol{v}^+\cdot\boldsymbol{n}^-\phi_h^+\psi^-\ ds = 0.$$

If the advection vector points into the cell K^+, figure A-1b and c, $\int_E \boldsymbol{v}^+\cdot\boldsymbol{n}^+\phi_h^+\psi^+\ ds =$

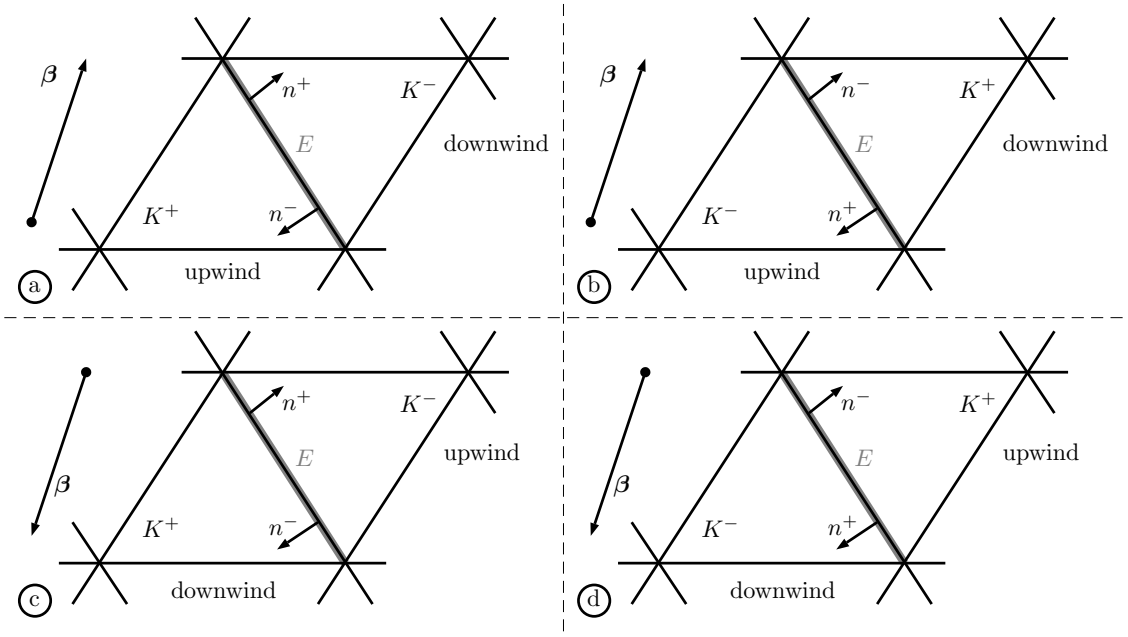

Figure A-1: The four possible cases for cell labelling and advection direction

0. And

$$\int_E \boldsymbol{v}^+ \cdot \boldsymbol{n}^- \phi_h^+ \psi^- \, ds = \int_E \boldsymbol{\beta} \cdot \boldsymbol{n}^- \phi_h^+ \psi^- \, ds,$$

which is the correct flux contribution using the quantity ϕ_h^+ to the cell K^-.

A similar trick is possible using the vector

$$\boldsymbol{v}^- = \frac{1}{2}(\boldsymbol{\beta} + |\boldsymbol{\beta} \cdot \boldsymbol{n}| \cdot \boldsymbol{n}^-),$$

Which gives the correct flux contributions using the quantity ϕ_h^-, for both the cases where the advection vector points into the cell K^- and K^+, and zero otherwise.

So the total flux is given by:

$$\begin{aligned}
&\int_E \boldsymbol{v}^+ \cdot \boldsymbol{n}^+ \phi_h^+ \psi^+ \, ds + \int_E \boldsymbol{v}^+ \cdot \boldsymbol{n}^- \phi_h^+ \psi^- \, ds + \int_E \boldsymbol{v}^- \cdot \boldsymbol{n}^+ \phi_h^- \psi^+ \, ds + \int_E \boldsymbol{v}^- \cdot \boldsymbol{n}^- \phi_h^- \psi^- \, ds \\
&\quad = \int_E (\boldsymbol{v}^- \phi_h^- + \boldsymbol{v}^+ \phi_h^+) \cdot (\psi^- \boldsymbol{n}^- + \psi^+ \boldsymbol{n}^+) \, ds
\end{aligned}$$

Comparing this to the expression in the weak form in equation (3.11), we see that

$$\begin{aligned}
(\boldsymbol{\beta}\phi_h)^* &= \boldsymbol{v}^- \phi_h^- + \boldsymbol{v}^+ \phi_h^+ \\
&= \frac{1}{2}\left[(\boldsymbol{\beta} + |\boldsymbol{\beta} \cdot \boldsymbol{n}| \boldsymbol{n}^-)\phi_h^- + (\boldsymbol{\beta} + |\boldsymbol{\beta} \cdot \boldsymbol{n}| \boldsymbol{n}^+)\phi_h^+\right] \\
&= \boldsymbol{\beta}\left(\frac{\phi_h^- + \phi_h^+}{2}\right) + \frac{1}{2}|\boldsymbol{\beta} \cdot \boldsymbol{n}|(\phi_h^- \boldsymbol{n}^- + \phi_h^+ \boldsymbol{n}^+) \\
&= \{\{\boldsymbol{\beta}\phi_h\}\} + \frac{1}{2} \, |\boldsymbol{\beta} \cdot \boldsymbol{n}| \, [\![\phi_h]\!],
\end{aligned}$$

which is precisely equation (A.1).

A.2 Bases

In order to move from a continuous representation of a function to a DOF vector and back again, we write a function as a linear combination of basis functions, the coefficients of which define the DOFs. We wrote this in equation (3.5), but did so without specifying any basis.

The choice of basis functions may affect the accuracy of the approximate solution, depending on the number of points used and their location within a cell. For our work we will only consider nodal bases, as discussed in section 3.2.2.

To start with we work using a domain $K = [0, 1]$, function space $\mathcal{V} = P^k(K)$ (the space of polynomials up to degree k) and DOFs $\mathcal{L} = \{\ell_j : j = 0, \ldots, n-1\}$, where $\ell_j(e) = e(x_j)$ is just point evaluation. This specifies the (Ciarlet) finite element[16] $(K, \mathcal{V}, \mathcal{L})$.

A.2.1 Basis functions

We start by approximating a function $f : [0, 1] \to \mathbb{R}$ by picking n points $x_0, \ldots, x_{n-1}$ in the interval $[0, 1]$. We then use these points as the interpolation points of the function f.

To determine the unique degree k polynomial through $k + 1$ points $x_0, \ldots, x_k$ we use Lagrange interpolation. The Lagrange basis functions are given by

$$e_i(x) = \prod_{\substack{m=0,\ldots,k \\ m \neq i}} \frac{x - x_m}{x_i - x_m} \qquad \text{for } i = 0, \ldots, k$$

The Lagrange basis functions have the property $e_i(x_j) = \delta_{ij}$.

If we want a finite dimensional approximation of $f(x)$ we can take the unique polynomial $f_h(x)$:

$$f(x) \approx f_h(x) = \sum_{i=0}^{k} f_i e_i(x),$$

where $f_i = f(x_i)$ for $i = 0, \ldots, k$.

Figure A-2 shows an example of how we construct the finite dimensional approximation in practice. The first (top left) plot shows the function, $f(x)$, that we are trying to approximate in red. The second (top right) shows the choice of three interpolation points x_0, x_1, x_2 marked with crosses on the x-axis and the value of the function at those points, $f_0 = f(x_0)$, $f_1 = f(x_1)$, $f_2 = f(x_2)$, is indicated with a circle marker. These values, f_0, f_1, f_2, are what go into the DOF vector.

Since we are using point evaluation and a nodal basis, each of the interpolation points, x_0, x_1, x_2, has an associated basis function e_0, e_1, e_2. The third (bottom left) plot in figure A-2 shows each of the three (quadratic) basis functions, each scaled by f_i. By scaling the basis functions we see that $f_i e_i(x)$ meets the original function $f(x)$ at precisely the nodal point x_i and is 0 at the other nodal points.

The final (bottom right) plot shows the approximation $f_h(x) = \sum_{i=0}^{2} f_i e_i(x)$, in green lying on top of the original function $f(x)$.

An alternative piecewise polynomial interpolation scheme can be obtained by dividing the domain up into a number of cells. After doing so we can approximate the portion of the function that is supported by each cell in the same way we did in figure A-2. The plots in figure A-3 demonstrate this process for a more complicated function plotted in red in the top left and approximation plotted in green in the bottom right. In figure A-3 we have not used the same interpolation points as in figure A-2, we are free to choose these points and figure A-4 shows 3 common choices of node placement.

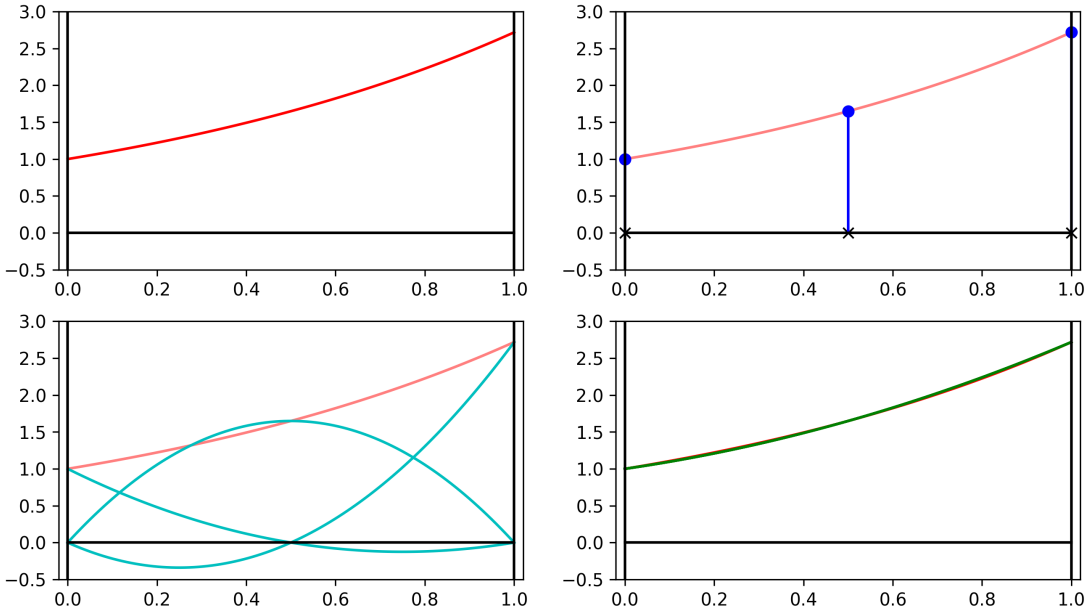

Figure A-2: Example of a finite dimensional approximation to a function using a degree 2 nodal basis

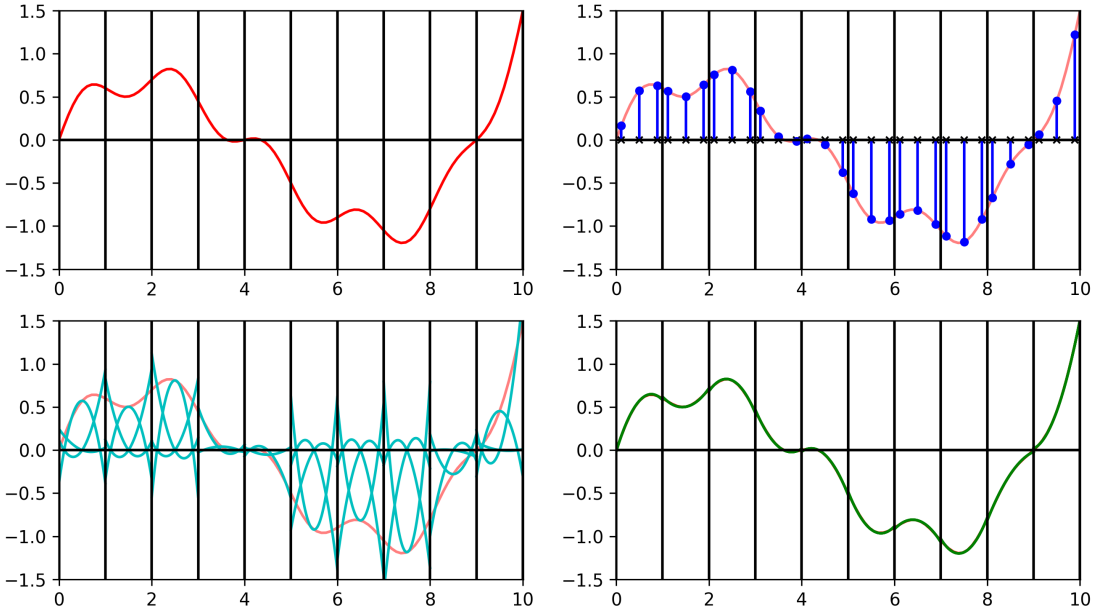

Figure A-3: Example of a finite dimensional approximation to a function using a degree 2 basis and Legendre nodal points

In figure A-2 we placed two interpolation points at the ends of the domain and one in the middle. But this is not the only choice for these points. Figure A-4 shows a three choices of interpolation points over the interval $[0, 1]$ for a range of different polynomial approximations. Also plotted in the same figure are the associated basis functions for each choice of points.

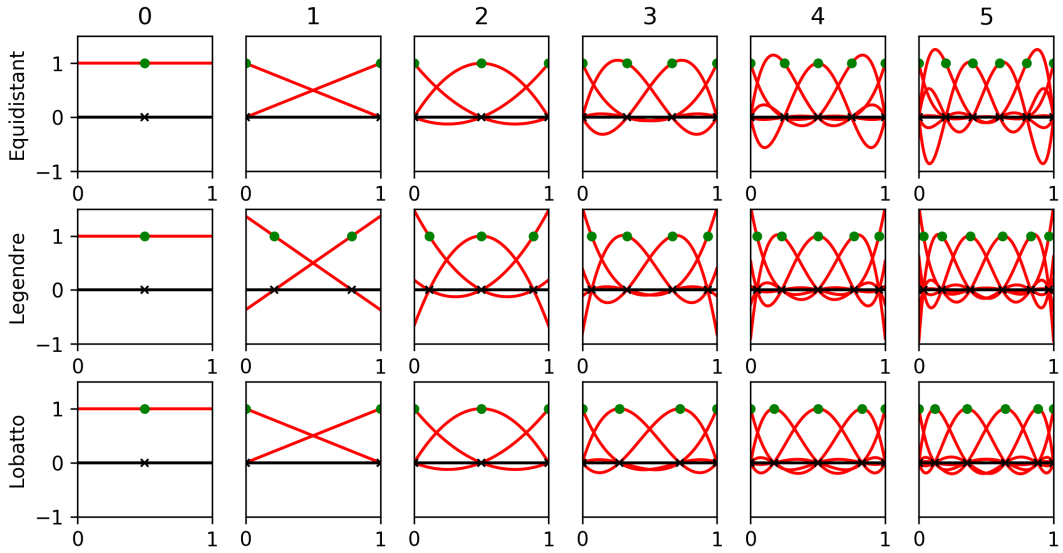

Figure A-4: Polynomial bases of degree 0-5 when using equidiatant, Legendre and Lobatto nodal points

The first row of plots in figure A-4 contains equidistant interpolation points. These are placed at the points $x_0 = 1/2$ for degree $k = 0$ and $x_i = i/k$ for $i = 0, \ldots, k-1$ for degree k polynomial basis functions. Notice that at higher degree, the red lines showing the shape of the basis function oscillate more near the end of the interval.

The second row of of plots in figure A-4 uses interpolation points at the Legendre nodes. These nodes are the zeros of the degree k Legendre polynomial, denoted $P_k(x)$, shifted from $[-1, 1]$ to the interval $[0, 1]$. Compared to the basis functions constructed using equidistant nodes, the basis functions at the ends of the interval do not oscillate as much. It is only at the very end of the interval that the basis functions begin to overshoot the value 1.

The third row of plots in figure A-4 uses interpolation points at the Lobatto nodes. These nodes are the zeros of the polynomial $(x^2 - 1)P'_{k-1}(x)$, were $P_k(x)$ is the Legendre polynomial (shifted from $[-1, 1]$ to $[0, 1]$). By including the two end points the basis functions do not overshoot 1 at the end of the interval. It is also advantageous to include the end points of the interval for a DG method as the function value at these points is stored in the DOF vector and can be used for the numerical flux calculations. If we do not include the end points of the interval, for instance when using Legendre nodes, in order to calculate the numerical flux we must evaluate the approximation $f_h(x)$ at the end of the interval to obtain the function value. For high degree polynomials or higher dimensions this can be a costly operation.

A.2.2 Differentiation and integration

Once we have specified the basis, the two key operations we must be able to perform on the approximate function f_h are differentiation and integration. Differentiation is

straightforward as the basis functions are just polynomials. For instance, we can write the x–derivative of f_h as

$$\partial_x f_h(x) = \sum_i f_h \partial_x e_i(x).$$

If we just store the coefficients of the polynomials e_i, then the exact derivative $\partial_x e_i(x)$ can be computed. Since the same basis functions are used in all cells, we can pre-compute all the derivatives of all the e_i we need and tabulate them.

Integration could also be performed exactly by using the coefficients of the polynomial basis functions, but in practise this is usually done through quadrature. When constructing the global system matrix we need to integrate expressions like $\int_0^1 e_i e_j \, dx$ to obtain each entry. If we did this calculation directly we would first have to find the coefficients of new polynomial $e_i(x)e_j(x)$ for each pair of basis functions. But by using Gauss quadrature we can transform the integration into a summation

$$\int_0^1 g(x) \, dx = \sum_\ell g(x_\ell)\omega_\ell + \varepsilon.$$

Here x_ℓ are a set of quadrature points in the interval $[0, 1]$, which have corresponding weights ω_ℓ. The value of ε depends on the regularity of the function g and choice of points x_ℓ. If we choose ℓ Legendre points and use the Gauss-Legendre quadrature rule, then polynomials up to degree $2\ell - 1$ can be integrated exactly. That is, if $g \in P^{2\ell-1}([0, 1])$, then $\varepsilon = 0$.

These ℓ Legendre quadrature points can correspond to the $k + 1$ interpolation points used to construct a degree k polynomial basis. If we consider $g(x) = e_i(x)e_j(x)$, where $e_i, e_j \in P^k([0, 1])$ and so $g \in P^{2k}([0, 1])$, then we require $2k < 2\ell - 1$ or $\ell > k + 1/2$ points for Gauss-Legendre to be exact.

There are also quadrature rules for other choices of points. For instance, if we were to choose Lobatto points, we can use the Gauss-Lobatto quadrature rule. However, this rule can only exactly integrate polynomials up to degree $2\ell - 3$, which necessitates using more quadrature points than a Lobatto basis is constructed on. We could choose to under-integrate these functions and accept this as another source of error, or we are free to use a combination of Lobatto interpolation points and Legendre quadrature points. In fact, there are even more choices of interpolation points and quadrature rules that could be used, but we won't cover these here.

A.2.3 Two dimensions

So far in this section we have only looked at approximating functions in one spatial dimension. Basis construction in two dimensions is more difficult.

To start we now have a large choice for the shape of our cells. We restrict our discussion to squares (quadrilaterals or quads) and triangles (simplices or tets) as these are most commonly used for DG methods.

We must then determine where to place the interpolation points within the cell. Unlike on an interval in one dimension, where $k + 1$ points determined a unique degree k polynomial in x, in two dimensions we require $\frac{1}{2}(k+1)(k+2)$ points to determine a degree k polynomial in x and y.

With a square we have the option of using a tensor product of the interpolation points

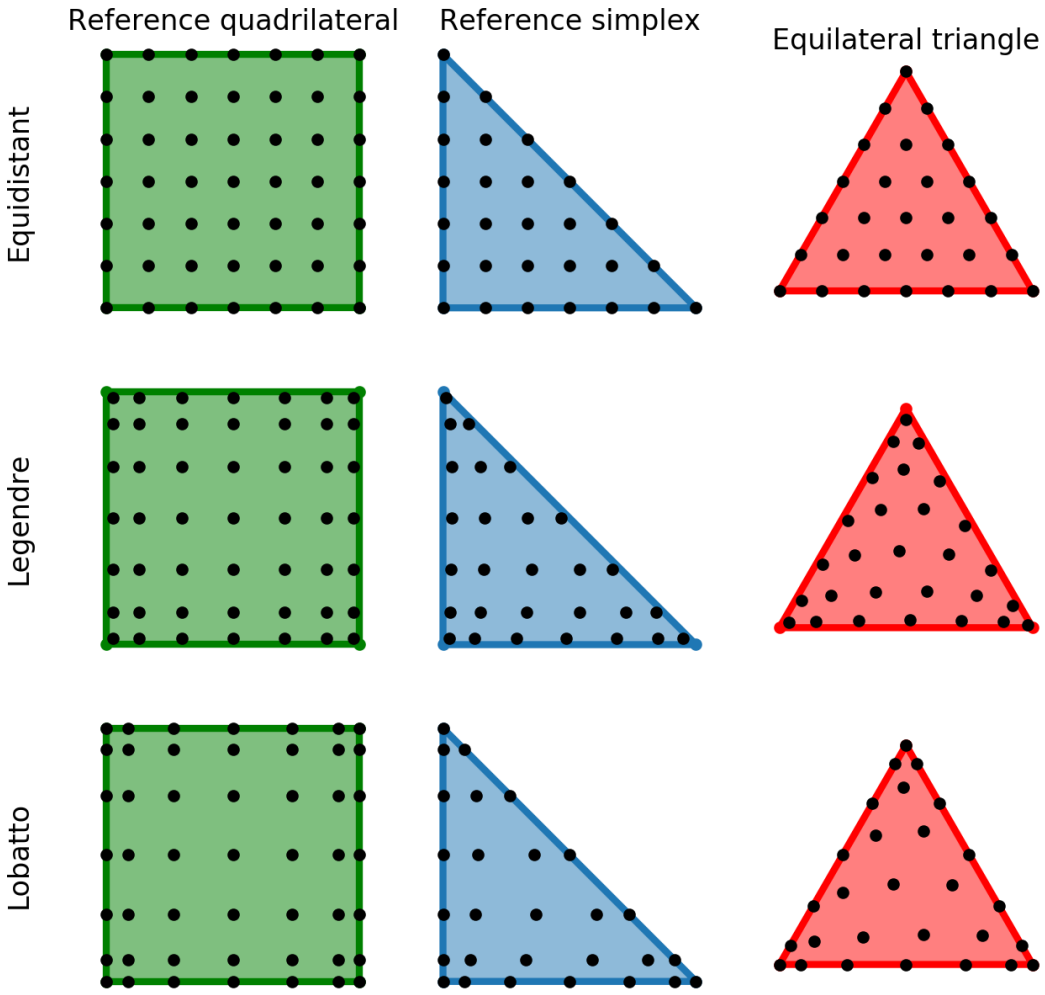

Figure A-5: Node placement for quadrilaterals and triangles in two dimensions using 7 points in each direction

we used previously. This means our approximate function is given by

$$f_h(x, y) = \sum_{i,j=0}^{k} f_{ij} e_i(x) e_j(y),$$

where $f_{ij} = f(x_i, y_j)$. If we do this, rather than having a degree k polynomial in x and y, the approximation is degree k in each spatial dimension, which is a degree $2k$ polynomial in x and y. Although this is not a total degree k approximation, tensor product basis functions are easier to construct and calculate with.

The first column of figure A-5 shows the placement of equidistant, Legendre and Lobatto points on a reference quadrilateral. The points are placed using the same positions as for one dimension in each of the x and y coordinates.

For a triangle we can take advantage of the tensor product if we remove some of the interpolation points. The second column of figure A-5, containing the reference sim-

plex,shows how this is achieved. The y coordinate is the same as for the reference quadrilateral in the first column. The x coordinate is determined using the $k+1$ equidistant/Legendre/Lobatto points on the bottom row of the triangle, the k points on the row above, reducing the number of points by 1 each time until the top row where there is only 1 point.

By reducing the number of points at each level there are exactly $\frac{1}{2}(k+1)(k+2)$ interpolation points, leading to a degree k polynomial approximation in x and y on a triangle.

This gives the finite dimensional approximation

$$f_h(x,y) = \sum_{j=0}^{k} \sum_{i=0}^{j} f_{ij} e_i^{(k-j)}(x) e_j^{(k)}(y),$$

where $e_i^{(j)}(x)$ denotes the i^{th} basis function on $j+1$ points in one dimension and $f_{ij} = f(x_i, y_j)$.

The disadvantage of this method of point placement on triangles is we not only have to construct the $k+1$ one dimensional interpolation points, but also the $k, \ldots, 1$ interpolation points too. Furthermore, the placement of interpolation points in this manner breaks the threefold symmetry of the triangle because points are mapped from the square to the triangle using a collapsed coordinate map. This is most readily seen in the third column of figure A-5, where the nodal points of the reference simplex are mapped to an equilateral triangle. A discussion of this map, and a technique to maintain triangular symmetry, are included in appendix A.2.4.

We will need a method to differentiate and integrate approximate functions constructed using a two dimensional basis. Differentiation depends on choice of basis, but if a tensor product basis is used we can calculate directional derivatives from the one dimensional components. For integration there are a plethora of quadrature rules in two dimensions. If a tensor product basis is used then on quadrilaterals Fubini's Theorem gives

$$\begin{aligned}\int_K e_i(x) e_j(y)\, d\boldsymbol{x} &= \int_{[0,1]} e_i(x)\, dx \int_{[0,1]} e_j(y)\, dy \\ &= \left(\sum_\ell e_i(x_\ell) w_\ell + \varepsilon_x \right) \left(\sum_m e_i(y_m) w_m + \varepsilon_y \right),\end{aligned}$$

where we have used quadrature rules in both the x and y directions independently.

In appendix A.2 the construction of basis functions in one spatial dimension is discussed. The construction in two spatial dimensions is much more difficult and figure A-5 is given as one example of how the interpolation points can be picked on quadrilaterals and triangles.

A.2.4 Point placement on simplices

The construction of points on the triangle is done using the collapsed coordinate map. All the points on a reference square with coordinates $(r,s) \in [-1,1] \times [-1,1]$ are mapped to the points on the reference simplex coordinates $(a,b) \in \{-1 \leq a, b \leq 1 : a+b \leq 0\}$ using the following formulae:

$$a = \frac{(1+r)(1-s)}{2} - 1, \quad b = s$$

The number of interpolation points is reduced on each row of the square so that there are the correct number on the triangle. This map works perfectly for equidistantly spaced interpolation, as the first row of shapes in figure A-5 shows. But in rows two and three there is clearly some distortion, which is most easily seen in the equilateral triangle as the breaking of three fold symmetry.

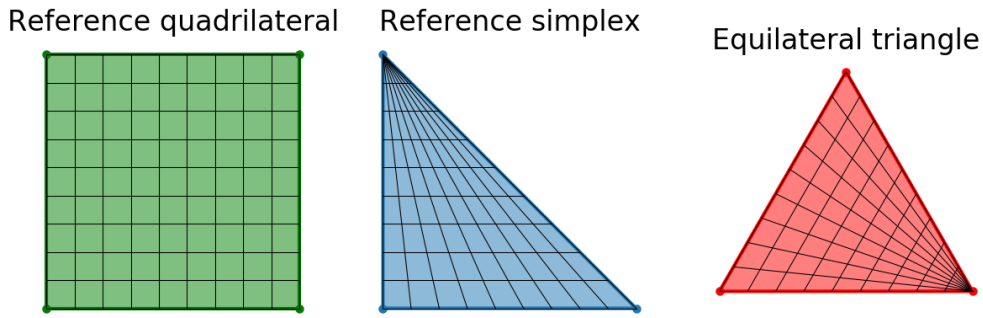

Figure A-6: Representation of collapsed coordinate map

The reason for this distortion is shown clearly in figure A-6, where the collapsed coordinate map has been applied to a grid on the reference quadrilateral and then mapped, first to the reference simplex and then to an equilateral triangle. One entire edge of the square gets contracted to a point, leaving grid lines clustered closer in one corner than the other two.

Figure A-7 shows a placement of interpolation points that retains three fold symmetry in the equilateral triangle and provides more even placement of points when using either Legendre or Lobatto placement.

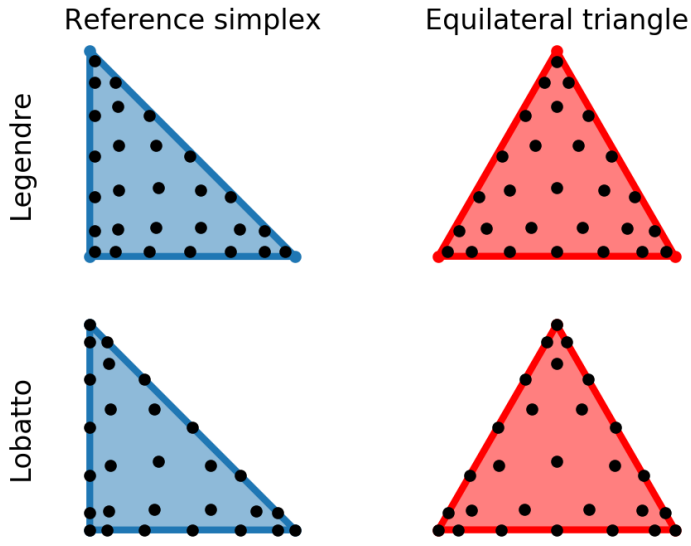

Figure A-7: Even placement using barycentric coordinates

Even with this placement, these are not the “optimal” points. For a thorough introduction to the construction of basis functions for different shapes using different point placement we suggest the book ‘Spectral/hp element methods for computational fluid

dynamics' by Karniadakis and Sherwin [35].

A.3 Explicit and Implicit Timestepping

A.3.1 Explicit Runge-Kutta

The two purely explicit methods we have seen so far are the explicit Euler method in section 2.4 and strong stability preserving Runge-Kutta order 3 (SSP RK3) method in section 3.2.3. We can generalise the concept of explicit timestepping methods by putting them into the Runge-Kutta framework.

An s stage explicit Runge-Kutta method for solving equation (4.11) is stepped forward in time using

$$q^{(n+1)} = q^{(n)} + \Delta t \sum_{i=1}^{s} b_i k_i, \tag{A.2}$$

where

$$k_i = f\left(t^{(n)} + c_i \Delta t, \quad q^{(n)} + \Delta t \sum_{j=1}^{i-1} a_{ij} k_j\right) \quad \text{for } i = 1, \ldots, s \tag{A.3}$$

where $c_1 = 0$ for the method to be explicit. Notice that the index of summation in equation (A.3) only goes up to $i - 1$, which allows for the calculation of the k_i using $k_1, \ldots, k_{i-1}$ calculated previously.

A convenient way of representing these coefficients is in a Butcher tableau. A generic Butcher tableau for an explicit method is shown in table A.1 and there are some examples in equation (A.4). Blank spaces in a Butcher tableau are zero coefficients. Notice that there are only coefficients below the leading diagonal, if there were any on or above the diagonal, the method would no longer be explicit.

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
$\vdots$	$\vdots$		$\ddots$		
c_s	a_{s1}	a_{s2}	$\cdots$	$a_{s\ s-1}$	
	b_1	b_2	$\cdots$	b_{s-1}	b_s

Table A.1: Form of a Butcher tableau for explicit RK timestepper

To actually perform the explicit Runge-Kutta timestepping we can use algorithm A.1 In this algorithm we specify the problem by providing f and an initial condition $q^{(0)}$. The algorithm then steps forward repeatedly by as step size that we specify (Δt) until we reach the final time T.

The algorithm is suitably generic that we can use any explicit Butcher tableau and it will use the associated method to timestep the problem. All the algorithms in this section and appendix A.3.2 and section 4.2.1 are written in this generic way, which allows us to change the timestepper by providing a different Butcher tableau as an input.

Algorithm A.1: Explicit Runge-Kutta

Input : $f : [0, T] \times \Omega \to \Omega$, $q^{(0)}$ – initial condition,
$(A, \underline{b}, \underline{c})$ – s-stage Butcher tableau,
T – Total time, Δt – Timestep size
Output: $q^{(n)}$ for $n = 0, \ldots, N = \lceil \frac{T}{\Delta t} \rceil$

$t = 0$
$n = 0$
while $t < T$ **do**
  $\tilde{q} = q^{(n)}$
  for $i = 1, \ldots, s$ **do**
    $k_i = f\left(t^{(n)} + c_i \Delta t, q^{(n)} + \Delta t \sum_{j=1}^{i-1} a_{ij} k_j\right)$
    $\tilde{q} = \tilde{q} + \Delta t b_i k_i$
  end
  $q^{(n+1)} = \tilde{q}$
  $t = t + \Delta t$
  $n = n + 1$
end
return $q^{(n)}, n = 0, \ldots, N$

$$
\begin{array}{c|c}
 & \\
0 & \\
\hline
 & 1
\end{array}
\qquad \text{Explicit Euler}
$$

$$
\begin{array}{c|cc}
0 & & \\
1 & 1 & \\
\hline
 & 1/2 & 1/2
\end{array}
\qquad \text{Heun}
$$

$$
\begin{array}{c|ccc}
0 & & & \\
1 & 1 & & \\
1/2 & 1/4 & 1/4 & \\
\hline
 & 1/6 & 1/6 & 2/3
\end{array}
\qquad \text{SSP RK3}
$$

$$
\begin{array}{c|ccccc}
0 & & & & & \\
1/2 & 1/2 & & & & \\
1/2 & 0 & 1/2 & & & \\
1 & 0 & 0 & 1 & & \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}
\qquad \text{“Classic” RK4}
\tag{A.4}
$$

Equation (A.4) gives four named examples of different explicit Runge-Kutta methods. The first and third we have seen before, the explicit Euler and strong stability preserving RK3 [49]. The second is Heun's method put into a Butcher tableau and the final method is the “classic” RK4 method, often referred to simply as *the* Runge-Kutta method.

Like other explicit methods, explicit Runge-Kutta methods are limited by the size of timestep that can be taken for the method to remain stable. In the next section we look at implicit Runge-Kutta methods, which allow for taking larger timesteps.

A.3.2 Implicit Runge-Kutta

To define an implicit Runge-Kutta method we again use equation (A.2) to calculate the next timestep, but calculate the k_i differently:

$$k_i = f\left(t^{(n)} + c_i\Delta t, \quad q^{(n)} + \Delta t \sum_{j=1}^{s} a_{ij}k_j\right) \quad \text{for } i = 1, \ldots, s. \tag{A.5}$$

Notice that the index of summation in equation (A.5) goes from 1 to s. This means we cannot directly calculate the k_i and must solve a system of equations to determine their value. For the general case in equation (4.11), if f is non-linear, this will be a system of non-linear equations.

Implicit Runge-Kutta methods can be represented using the full Butcher tableau as shown in table A.2. Some examples are also shown in equation (A.6).

c_1	a_{11}	a_{12}	$\cdots$	a_{1s}
c_2	a_{21}	a_{22}		a_{2s}
$\vdots$	$\vdots$		$\ddots$	$\vdots$
c_s	a_{s1}	a_{s2}	$\cdots$	a_{ss}
	b_1	b_2	$\cdots$	b_s

Table A.2: Form of a Butcher tableau for implicit RK timestepper

Algorithm A.2: Implicit Runge-Kutta

Input : $f : [0, T] \times \Omega \to \Omega$, $q^{(0)}$ – initial condition,
$(A, \underline{b}, \underline{c})$ – s-stage Butcher tableau,
T – Total time, Δt – Timestep size
Output: $q^{(n)}$ for $n = 0, \ldots, N = \lceil \frac{T}{\Delta t} \rceil$

$t = 0$
$n = 0$
while $t < T$ **do**
  $F_i(k_i) := f\left(t^{(n)} + c_i\Delta t, q^{(n)} + \Delta t \sum_{j=1}^{s} a_{ij}k_j\right)$
  $\underline{k} = \text{Solve}\Big(F(\tilde{\underline{k}}; t, q) = \tilde{\underline{k}}\Big)$ – Expensive
  $q^{(n+1)} = q^{(n)} + \sum_{i=1}^{s} b_i k_i$
  $t = t + \Delta t$
  $n = n + 1$
end
return $q^{(n)}, n = 0, \ldots, N$

Algorithm A.2 show how the implicit Runge-Kutta method can be implemented. Unlike the explicit Runge-Kutta algorithm, algorithm A.1, there is no for loop. This is replaced by a solve step instead, which is very expensive. We have made no assumptions on f and this could be a complicated non-linear function that is expensive to evaluate. If this is the case the solve step is a non-linear solve which requires a great deal of computational effort to complete. It is clear that this solve is considerably more expensive that the evaluation that is performed in algorithm A.1.

For the generic problem we study in this chapter, we do not specify whether q is a

scalar or vector, since the timestepping methods work in both cases. It is possible that the function f is linear, like equation (3.45) that we consider in section 3.3.3, and can be written as a matrix equation $f(\underline{q}) = A\underline{q}$. Even if this is the case, the solve step is still very expensive as the linear system that has to be solved is s times the size of the matrix A, where s is then number of stages in the Runge-Kutta method. That is if $A \in \mathbb{R}^{m\times m}$, then the matrix in the solve step is in $\mathbb{R}^{sm\times sm}$.

Less expensive implicit Runge-Kutta methods use only coefficients on the diagonal of the Butcher tableau and not above. This replaces the one big solve using a matrix in $\mathbb{R}^{sm\times sm}$ with s solves using a matrix in $\mathbb{R}^{m\times m}$. These methods are called Diagonally Implicit Runge-Kutta (DIRK) methods, and are what we will use in our implementation.

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \qquad \begin{array}{c|cc} 0 & & \\ 1 & (1-\theta) & \theta \\ \hline & (1-\theta) & \theta \end{array} \tag{A.6}$$

Implicit Euler Theta Method

The methods in equation (A.6) are both DIRK methods that we have seen previously, although not in the Runge-Kutta framework. The first is the implicit Euler method and the second is the theta method. Both only have entries on or below the leading diagonal.

A.4 Patch preconditioner

From the discussion in chapter 6, we stated that the coarse space problem for the Lax-Friedrichs flux is

$$a_0(\boldsymbol{q}, \boldsymbol{p}) = -\left(\frac{\phi_B}{\rho_1}\nabla\nabla\cdot\boldsymbol{q}, \boldsymbol{p}\right) + (\rho_2\boldsymbol{q}, \boldsymbol{p}). \tag{A.7}$$

The difficulty encountered when trying to solve a system based on this bilinear form is that the gradient of the divergence differential operator $\nabla\nabla\cdot(\boldsymbol{q})$ has a null space. Consider a vector valued function $\boldsymbol{q}$ which is linear in both components, then $\nabla\nabla\cdot(\boldsymbol{q}) = \boldsymbol{0}$.

To numerically solve problems with a kernel we can use a patch preconditioner (`PATCHPC`). A similar technique is used in Farrell et al. [27] to stabilise the grad-div operator for the Stokes problem. An example of PETSc options to use PatchPC are included in listing A.1.

```
# GMG paramters for Lax-Friedrichs flux
mg_param = {'ksp_type':'cg',
            'mat_type':'aij',
            'pc_type': 'mg',
            'pc_mg_type': 'full',
            'pc_mg_cycles': 'v',
            'mg_levels': {'ksp_type': 'richardson',
                          'ksp_norm_type': 'unpreconditioned',
                          'ksp_richardson_scale': 0.5,
                          'ksp_max_it': 1,
                          'ksp_convergence_test': 'skip',
                          'pc_type': 'python',
                          'pc_python_type': 'firedrake.PatchPC',
                          'patch':
                              {'pc_patch':
                                    {'save_operators': True,
                                     'partition_of_unity': False,
                                     'construct_type': 'star',
                                     'construct_dim': 0,
                                     'sub_mat_type': 'seqdense',
                                     'precompute_element_tensors': True},
                                'sub_ksp_type': 'preonly',
                                'sub_pc_type': 'lu'}},
            'mg_coarse': {'pc_type': 'python',
                          'pc_python_type': 'firedrake.AssembledPC',
                          'assembled_pc_type': 'lu',
                          'assembled_pc_factor_mat_solver_type': 'mumps'},
            'ksp_monitor_true_residual':None,
            'ksp_rtol': 1E-8}
```

Listing A.1: Example code

It is possible to solve the grad–div problem shown in equation (A.7) using Firedrake in using an independent script. However, it is not possible at this time to compose this set of PETSc solver options with the `firedrake.GTMGPC` preconditioner when solving the SWE, like we did for the upwind flux in section 7.6.3.

BIBLIOGRAPHY

[1] FEniCS website. https://fenicsproject.org/. Accessed: 2019-10-16.

[2] Firedrake website. http://www.firedrakeproject.org/. Accessed: 2019-10-06.

[3] Hurricane season 2005: Katrina. https://www.nasa.gov/vision/earth/lookingatearth/h2005_katrina_prt.htm. Accessed: 2019-10-20.

[4] Met Office numerical weather prediction models. https://www.metoffice.gov.uk/research/modelling-systems/unified-model/weather-forecasting. Accessed: 2019-10-20.

[5] NASA images of hurricane Katrina and 2005 storm season available. https://www.nasa.gov/press-release/goddard/nasa-images-of-hurricane-katrina-and-2005-storm-season-available. Accessed: 2019-10-20.

[6] J. Ahrens, B. Geveci, C. Law, C. Hansen, and C. Johnson. 36-ParaView: An end-user tool for large-data visualization, 2005.

[7] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software (TOMS)*, 40(2):9, 2014.

[8] U. M. Ascher, S. J. Ruuth, and R. J. Spiteri. Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25(2-3):151–167, 1997.

[9] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.12, Argonne National Laboratory, 2019.

[10] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc Web page. https://www.mcs.anl.gov/petsc, 2019.

[11] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[12] P. Bastian, E. H. Müller, S. Müthing, and M. Piatkowski. Matrix-free multigrid block-preconditioners for higher order discontinuous Galerkin discretisations. *Journal of Computational Physics*, 2019.

[13] M. Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[14] W. L. Briggs, S. F. McCormick, et al. *A multigrid tutorial*. Siam, 2000.

[15] T. Bui-Thanh. Construction and analysis of HDG methods for linearized shallow water equations. *SIAM Journal on Scientific Computing*, 38(6):A3696–A3719, 2016.

[16] P. G. Ciarlet. *The finite element method for elliptic problems*, volume 40. Siam, 2002.

[17] B. Cockburn. Discontinuous galerkin methods. *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, 83(11):731–754, 2003.

[18] B. Cockburn, B. Dong, and J. Guzmán. Optimal convergence of the original DG method for the transport-reaction equation on special meshes. *SIAM Journal on Numerical Analysis*, 46(3):1250–1265, 2008.

[19] B. Cockburn, O. Dubois, J. Gopalakrishnan, and S. Tan. Multigrid for an HDG method. *IMA Journal of Numerical Analysis*, 34(4):1386–1425, 2014.

[20] B. Cockburn and C.-W. Shu. The local discontinuous Galerkin method for time-dependent convection-diffusion systems. *SIAM Journal on Numerical Analysis*, 35(6):2440–2463, 1998.

[21] B. Cockburn and C.-W. Shu. Runge–Kutta discontinuous Galerkin methods for convection-dominated problems. *Journal of scientific computing*, 16(3):173–261, 2001.

[22] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM journal of Research and Development*, 11(2):215–234, 1967.

[23] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo. Parallel distributed computing using Python. *Advances in Water Resources*, 34(9):1124–1139, 2011.

[24] A. Dedner, E. Müller, and R. Scheichl. Efficient multigrid preconditioners for atmospheric flow simulations at high aspect ratio. *International Journal for Numerical Methods in Fluids*, 80(1):76–102, 2016.

[25] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[26] H. C. Elman, D. J. Silvester, and A. J. Wathen. *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. Oxford University Press, USA, 2014.

[27] P. E. Farrell, L. Mitchell, and F. Wechsung. An augmented Lagrangian preconditioner for the 3D stationary incompressible Navier-Stokes equations at high Reynolds number. *arXiv preprint arXiv:1810.03315*, 2018.

[28] B. Fraeijs de Veubeke. Displacement and equilibrium models in the finite element method. *Stress analysis*, pages chapter–9, 1965.

[29] T. H. Gibson, L. Mitchell, D. A. Ham, and C. J. Cotter. A domain-specific language for the hybridization and static condensation of finite element methods. *arXiv preprint arXiv:1802.00303*, 2018.

[30] F. X. Giraldo and M. Restelli. High-order semi-implicit time-integrators for a triangular discontinuous Galerkin oceanic shallow water model. *International journal for numerical methods in fluids*, 63(9):1077–1102, 2010.

[31] J. Gopalakrishnan and S. Tan. A convergent multigrid cycle for the hybridized mixed method. *Numerical Linear Algebra with Applications*, 16(9):689–714, 2009.

[32] M. Homolya, L. Mitchell, F. Luporini, and D. A. Ham. TSFC: a structure-preserving form compiler. *SIAM Journal on Scientific Computing*, 40(3):C401–C428, 2018.

[33] C. Johnson and J. Pitkäranta. An analysis of the discontinuous Galerkin method for a scalar hyperbolic equation. *Mathematics of computation*, 46(173):1–26, 1986.

[34] S. Kang, F. X. Giraldo, and T. Bui-Thanh. IMEX HDG-DG: a coupled implicit hybridized discontinuous Galerkin (HDG) and explicit discontinuous Galerkin (DG) approach for shallow water systems. *arXiv preprint arXiv:1711.02751*, 2017.

[35] G. Karniadakis and S. Sherwin. *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, 2013.

[36] A. Klöckner. Loo.py: transformation-based code generation for GPUs and CPUs. In *Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming*, Edinburgh, Scotland., 2014. Association for Computing Machinery.

[37] P. H. Lauritzen, C. Jablonowski, M. A. Taylor, and R. D. Nair. *Numerical techniques for global atmospheric models*, volume 80. Springer Science & Business Media, 2011.

[38] P. Lesaint and P. Raviart. On a finite element method for solving the neutron transport equation. *Publications mathématiques et informatique de Rennes*, (S4):1–40, 1974.

[39] A. Logg, K. Mardal, and G. Wells. The fenics book, 2011.

[40] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM journal on numerical analysis*, 12(4):617–629, 1975.

[41] L. Pareschi and G. Russo. Implicit–explicit Runge–Kutta schemes and applications to hyperbolic systems with relaxation. *Journal of Scientific computing*, 25(1):129–155, 2005.

[42] J. Peraire, N. Nguyen, and B. Cockburn. A hybridizable discontinuous Galerkin method for the compressible Euler and Navier-Stokes equations. In *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, page 363, 2010.

[43] J. Qiu, B. C. Khoo, and C.-W. Shu. A numerical study for the performance of the runge–kutta discontinuous galerkin method based on different numerical fluxes. *Journal of Computational Physics*, 212(2):540–565, 2006.

[44] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. Kelly. Firedrake: automating the finite element method by composing abstractions. *arXiv preprint arXiv:1501.01809*, 2015.

[45] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. Kelly. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1116–1123. IEEE, 2012.

[46] W. H. Reed and T. Hill. Triangular mesh methods for the neutron transport equation. Technical report, Los Alamos Scientific Lab., N. Mex.(USA), 1973.

[47] C. Rossby. Planetary flow patterns in the atmosphere. *Quart. J. Roy. Met. Soc*, 66:68, 1939.

[48] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[49] C.-W. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *Journal of computational physics*, 77(2):439–471, 1988.

[50] A. Staniforth and J. Thuburn. Horizontal grids for global weather and climate prediction models: a review. *Quarterly Journal of the Royal Meteorological Society*, 138(662):1–26, 2012.

[51] A. Stuart and J. Voss. Matrix analysis and algorithms, 2009.

[52] K. Stüben. A review of algebraic multigrid. In *Numerical Analysis: Historical Developments in the 20th Century*, pages 331–359. Elsevier, 2001.

[53] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. Kelly. A study of vectorization for matrix-free finite element methods. *arXiv preprint arXiv:1903.08243*, 2019.

[54] U. Trottenberg, C. W. Oosterlee, and A. Schuller. *Multigrid*. Elsevier, 2000.

[55] H. Weller, S.-J. Lock, and N. Wood. Runge–Kutta IMEX schemes for the horizontally explicit/vertically implicit (HEVI) solution of wave equations. *Journal of Computational Physics*, 252:365–381, 2013.

[56] T. Wildey, S. Muralikrishnan, and T. Bui-Thanh. Unified geometric multigrid algorithm for hybridized high-order finite element methods. *arXiv preprint arXiv:1811.09909*, 2018.

[57] D. L. Williamson, J. B. Drake, J. J. Hack, R. Jakob, and P. N. Swarztrauber. A standard test set for numerical approximations to the shallow water equations in spherical geometry. *Journal of Computational Physics*, 102(1):211–224, 1992.